UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik

Parallele Kürzung von Rang-k-Matrizen

Bachelorarbeit

Leipzig, Februar, 2009

vorgelegt von
Drechsler, Florian
geb. am: 20.03.1980
Studiengang Informatik

Zusammenfassung

Die Kürzung einer Rang-k-Matrix ist ein wichtiger Bestandteil der Technik der Hierarchischen Matrizen. In dieser Arbeit untersuchen wir zwei verschiedene Kürzungsalgorithmen auf ihre Parallelisierbarkeit.
Zuerst werden wir die sequentiellen Versionen der Algorithmen einführen, ihre Komplexität untersuchen und diese Ergebnisse in numerischen Experimenten validieren.
Danach parallelisieren wir beide Algorithmen und untersuchen ihr Laufzeitverhalten theoretisch und anhand von numerischen Experimenten auf Rechensystemen mit verteiltem und geteiltem Speicher. Es zeigt sich, dass beide Algorithmen gut parallelisierbar sind, wobei wir bei Rechensystemen mit verteiltem Speicher die Anzahl der verwendeten Prozessoren an die Größe der zu kürzenden Matrix anpassen sollten, damit wir einen linearen Speedup erreichen.

Danksagung

Ich bin sehr dankbar, dass ich diese Arbeit am Max-Planck-Institut für Mathematik in den Naturwissenschaften vollenden konnte. Besonders danke ich Dr. Ronald Kriemann für das Thema dieser Arbeit, die aufgewendete Geduld sowie für die vielen hilfreichen Tipps und Diskussionen.

Des Weiteren danke ich:
Prof. Dr. M. Middendorf für die Betreuung dieser Arbeit an der Universität Leipzig.
Prof. Dr. Dr. h.c. W. Hackbusch für Betreuung dieser Arbeit am Max-Planck-Institut für Mathematik in den Naturwissenschaften.
Stefan Kühn, Lars Grasedyck, Konrad Kaltenbach, Jan Schneider, Ricardo Reiche, meiner Familie und allen Freunden für die motivierenden Worte bei der Bearbeitung dieses Themas.

Inhaltsverzeichnis

Liste der Algorithmen

Kapitel 1

Einleitung

Wir können Probleme der Physik mittels Integralgleichungen oder partiellen Differentialgleichungen beschreiben. Falls wir keine analytische Lösung dieser Gleichung angeben können, so haben wir die Möglichkeit, eine Lösung mittels eines numerischen Verfahrens zu approximieren. Diskretisieren wir zum Beispiel eine Integralgleichung, so erhalten wir ein Gleichungssystem

$$Ax = b, \quad A \in \mathbb{R}^{n \times n}, \quad x, b \in \mathbb{R}^n. \tag{1.1}$$

Die Matrix A ist in der Regel vollbesetzt ([31]), wodurch der Speicheraufwand der Matrix quadratisch von der Größe n abhängt.
Wir können die Komplexität senken, indem wir Hierarchische Matrizen verwenden ([18]). In [18, 19] wird gezeigt, dass wir Teilbereiche der Matrix A durch Niedrigrangmatrizen approximieren können. Die Hierarchischen Matrizen verwenden Rang-k-Matrizen, um diese Niedrigrangmatrizen zu speichern. Die Komplexität des Speicherbedarfs ist $\mathcal{O}(kn \log n)$, somit hat die Größe k der Rang-k-Matrizen Einfluss auf den Speicherbedarf. Zusätzlich lässt sich der Approximationsfehler durch den Parameter k der Rang-k-Matrizen steuern.
Verschiedene Methoden zum Aufstellen einer Hierarchischen Matrix oder auch die Addition zweier Hierarchischer Matrizen führen zu einem zu großen k.
In diesen Fällen ist es nötig, k zu verkleinern. Dies wird mit der Rang-k-Matrix-Kürzung realisiert. Die sequentielle Kürzung wurde zum Beispiel in [18, 19] behandelt. Damit wir Hierarchische Matrizen auch auf Computersystemen mit verteiltem Speicher verwenden können, benötigen wir die Kürzung der Rang-k-Matrizen ebenfalls in einer parallelen Version. Diese Version wollen wir in dieser Arbeit definieren und analysieren.
Dazu werden wir in diesem Kapitel die Rang-k-Matrizen und die Eigenschaft der Kürzung einführen. In Kapitel 2 definieren wir zwei sequentielle Algorithmen, um die Kürzung durchzuführen. Der erste Algorithmus verwendet die Singulärwertzerlegung, der zweite Algorithmus löst ein Eigenwertproblem. Wir geben die Algorithmen in Pseudocode an, untersuchen die Komplexität und validieren die Komplexitätsabschätzung anhand von Experimenten.
Die Algorithmen aus Kapitel 2 parallelisieren wir in Kapitel 3 und untersuchen die Laufzeit der parallelen Algorithmen im Vergleich zu den sequentiellen Algorithmen.

Bevor wir die Rang-k-Matrizen vorstellen, wollen wir zuerst noch eine hilfreiche Notation einführen.

1.1 Die Doppelpunkt-Notation

In [17] wird die Doppelpunkt-Notation (engl. „colon notation“) eingeführt. Mithilfe dieser Notation können wir Spalten, Zeilen oder Blöcke einer Matrix leicht spezifizieren. Wir verwenden diese Notation, weil wir die Algorithmen dadurch kompakter und verständlicher aufschreiben können.

Notation 1.1.1 *Sei $A \in \mathbb{R}^{n\times m}$ eine Matrix.*
Wir bezeichnen das Element A_{ij} der Matrix in Zukunft mit

$$A(i,j) := A_{ij}, \quad i \in \{1,\dots,n\},\ j \in \{1,\dots m\}. \tag{1.2}$$

Wir geben die i-te Zeile der Matrix A mittels des Platzhalters : folgendermaßen an:

$$A(i,:) := [a_{i1},\dots,a_{im}] \in \mathbb{R}^{1\times m}. \tag{1.3}$$

Die j-te Spalte geben wir analog an:

$$A(:,j) := \begin{bmatrix} a_{1j} \\ a_{2j} \\ \cdot \\ \cdot \\ \cdot \\ a_{nj} \end{bmatrix} \in \mathbb{R}^{n\times 1}. \tag{1.4}$$

Für spätere Anwendungen benötigen wir Teile einer Spalte oder Zeile:

$$A(i,\sigma:\tau) := [a_{i\sigma},\dots,a_{i\tau}] \in \mathbb{R}^{1\times \tau-\sigma+1} \tag{1.5}$$

und

$$A(\sigma:\tau,j) := \begin{bmatrix} a_{\sigma j} \\ a_{\sigma+1 j} \\ \cdot \\ \cdot \\ \cdot \\ a_{\tau j} \end{bmatrix} \in \mathbb{R}^{\tau-\sigma+1\times 1}. \tag{1.6}$$

Die Bestimmung von Untermatrizen der Matrix A ergibt sich nun durch

$$A(i:j,\sigma:\tau) := \begin{bmatrix} a_{i\sigma} & a_{i(\sigma+1)} & \cdots & a_{i\tau} \\ a_{(i+1)\sigma} & a_{(i+1)(\sigma+1)} & \cdots & a_{(i+1)\tau} \\ \cdots & \cdots & \ddots & \cdots \\ a_{j\sigma} & a_{j(\sigma+1)} & \cdots & a_{j\tau} \end{bmatrix} \in \mathbb{R}^{(j-i+1)\times(\tau-\sigma+1)}. \tag{1.7}$$

Im Falle von Vektoren $v \in \mathbb{R}^n$ sparen wir die Spezifikation der Spalten, und wir erhalten für die Auswahl eines Teils des Vektors folgende Notation:

$$v(i:j) := \begin{bmatrix} v_i \\ v_{i+1} \\ \cdot \\ \cdot \\ \cdot \\ v_j \end{bmatrix} \in \mathbb{R}^{j-i+1}. \tag{1.8}$$

1.2 Rang-k-Matrizen

Nun wollen wir die Rang-k-Matrizen einführen und ihre Eigenschaften untersuchen.

1.2.1 Die Rang-k-Matrizen und ihre Eigenschaften

Zuerst definieren wir die Menge der $(n \times m)$-Matrizen mit maximalem Rang k.

Definition 1.2.1 (Die Menge der Niedrigrangmatrizen [18]) *Sei $n, m \in \mathbb{N}$, $k \in \mathbb{N}_0$. Wir definieren die Menge der $(n \times m)$-Matrizen mit einem maximalen Rang k durch*

$$\mathcal{R}_{\leq k}(n,m) := \{M \in \mathbb{R}^{n \times m} \mid rang(M) \leq k\}. \tag{1.9}$$

Damit der Speicherverbrauch sinkt, werden Niedrigrangmatrizen nicht als vollbesetzte Matrizen abgespeichert, sondern als Produkt von zwei kleinen Matrizen.

Definition 1.2.2 (Rang-k-Matrizen [18]) *Eine Matrix $M \in \mathcal{R}_{\leq k}(n,m)$ bezeichnen wir als Rang-k-Matrix, wenn sie in folgender Form vorliegt:*

$$M = AB^T, \quad A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{m \times k}. \tag{1.10}$$

Die Menge aller $(n \times m)$-Rang-k-Matrizen bezeichnen wir mit $\mathcal{R}k(n,m)$.

Im Folgenden werden wir nur mit Rang-k-Matrizen arbeiten, dass heißt, falls wir eine Rang-k-Matrix vorliegen haben, werden wir sie gleich in der faktorisierten Form verwenden.

Notation 1.2.3 (Speicherbedarf und Komplexität) *In dieser Arbeit betrachten wir den Speicherbedarf von Strukturen, sowie den Rechenaufwand von Algorithmen. Den Speicherbedarf einer Struktur geben wir durch*

$$\mathcal{S}_{Str}(\cdot) \tag{1.11}$$

an. Dabei ist Str die Bezeichnung oder eine Abkürzung der Struktur, und in den Klammern stehen die Parameter, von denen die Struktur abhängt. Im Falle

einer Rang-k-Matrix $M = AB^T \in \mathbb{R}^{n\times m}$ sind dies n, m und k. Den Rechenaufwand eines Algorithmus geben wir mit

$$\mathcal{N}_{Alg}(\cdot) \tag{1.12}$$

an, wobei Alg die Abkürzung für den Algorithmus ist. In den Klammern stehen wieder die Parameter, von denen der Algorithmus abhängt.

Wird zum Beispiel bei einer Matrix $M \in \mathbb{R}^{n\times m}$ jeder Eintrag abgespeichert, so erhalten wir für den Speicherbedarf

$$\mathcal{S}_V(n, m) = nm. \tag{1.13}$$

V ist in diesem Fall die Abkürzung für eine vollbesetzte Matrix.

Bemerkung 1.2.4 (Speicherbedarf [15, 18]) *Der Speicherbedarf einer Rang-k-Matrix $R = AB^T \in \mathbb{R}^{n\times m}$ beträgt*

$$\mathcal{S}_{Rk}(R) = nk + mk. \tag{1.14}$$

Wir können den Speicherbedarf weiter abschätzen, falls $m \leq n$ gilt:

$$\mathcal{S}_{Rk}(R) \leq 2nk = \mathcal{O}(nk). \tag{1.15}$$

Wenn $n, m \gg k$ gilt, ist der Speicherbedarf einer im Rang-k-Matrix-Format abgespeicherten Matrix geringer, verglichen mit einer $(n \times m)$-Matrix, die im vollbesetzten Format gespeichert wurde.

Rang-k-Matrizen werden in der Regel dazu verwendet, um vollbesetzte Matrizen zu approximieren.
Ein weiterer Vorteil neben der Reduzierung des Speicherbedarfs durch Rang-k-Matrizen ist, dass Matrixoperationen wie die Matrix-Vektor-Multiplikation billiger sind.

Bemerkung 1.2.5 (Matrix-Vektor-Multiplikation [15]) *Die Matrix-Vektor-Multiplikation mit einer Rang-k-Matrix $M = AB^T \in \mathbb{R}^{n\times m}$ kostet*

$$\mathcal{N}_{MVRk}(n, m, k) = 2k(n + m) - n - k \leq 2k(n + m). \tag{1.16}$$

Die Rang-k-Matrix-Kürzung hat nun das Ziel, eine Rang-k-Matrix $R = AB^T \in \mathcal{R}k(n, m)$ in eine Matrix $R' = A'B'^T \in \mathcal{R}k'(n, m)$, $k' < k$, zu überführen, sodass ein vorgegebener Approximationsfehler $\epsilon \geq 0$

$$\|R - R'\|_2 \leq \epsilon \tag{1.17}$$

eingehalten wird. Die Matrix R' wird als Rang-k'-Matrix gespeichert.

1.2.2 Kürzung mittels Bestapproximation

Es ist möglich, jede Matrix mittels Bestapproximation bezüglich der Spektral- oder Frobeniusnorm in eine Rang-k-Matrix-Approximation zu überführen. Für diese Operation benötigen wir die Singulärwertzerlegung. Der folgende Teil wurde grösstenteils aus [15] übernommen.

Singulärwertzerlegung

Satz 1.2.6 (Singulärwertzerlegung von Matrizen) *Sei $M \in \mathbb{R}^{n\times m}$ gegeben, dann existieren orthogonale Matrizen $U \in \mathbb{R}^{n\times n}$ und $V \in \mathbb{R}^{m\times m}$, sodass*

$$M = U\Sigma V^T \text{ mit } \Sigma \in \mathbb{R}^{n\times m} \tag{1.18}$$

gilt. Hierbei ist

$$\Sigma = diag(\sigma_1, \ldots, \sigma_p), \tag{1.19}$$

eine rechteckige Diagonalmatrix mit $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$, $p = \min\{n, m\}$.

Die Singulärwertzerlegung hat folgende Eigenschaften für die Frobenius- und Spektralnorm.

Satz 1.2.7 *Es seien die Matrix $M \in \mathbb{R}^{n\times m}$ und ihre Singulärwertzerlegung durch $M = U\Sigma V^T$ nach (1.18) gegeben. Dann sind die Werte der Spektral- und Frobeniusnorm durch*

$$\|M\|_2 = \sigma_1 \quad und \quad \|M\|_F = \sqrt{\sum_{i=1}^{\min\{n,m\}} \sigma_i^2} \tag{1.20}$$

bestimmt.

Beweis: Der Leser findet den Beweis in [21, Lemma C.2.3].

■

Mittels der Singulärwertzerlegung können wir eine Bestapproximation für einen festen vorgegebenen Rang bezüglich der Frobenius- und Spektralnorm angeben.

Satz 1.2.8 (Bestapproximation mittels Niedrigrangmatrizen) *Sei $M \in \mathbb{R}^{n\times m}$ mit der Singulärwertzerlegung $M = U\Sigma V^T$ wie in Satz 1.2.6 gegeben. Die Minimierungsprobleme*

$$\min_{Rang(R)\leq k} \|M - R\|_2 \quad und \quad \min_{Rang(R)\leq k} \|M - R\|_F \tag{1.21}$$

werden durch

$$R := U\Sigma_k V^T \text{ mit } (\Sigma_k)_{ij} = \begin{cases} \sigma_i & : \text{ falls } i = j \leq \min\{k, n, m\} \\ 0 & : \text{ sonst} \end{cases}$$

gelöst. Der Approximationsfehler bezüglich der beiden Normen ist

$$\|M - R\|_2 = \sigma_{k+1} \quad bzw. \quad \|M - R\|_F = \sqrt{\sum_{i=k+1}^{\min\{n,m\}} \sigma_i^2}. \tag{1.22}$$

Beweis: Der Beweis findet sich in [21, Lemma C.2.3 und C.2.4].

■

Damit haben wir die Werkzeuge, um eine vollbesetzte Matrix durch eine Niedrigrangmatrix zu approximieren. Wir müssen aber beachten, dass die Matrix R aus Satz 1.2.8 nicht eindeutig bestimmt ist. Die Bestapproximation mittels Rang-k-Matrizen gelingt nun mit der sogenannten komprimierten Singulärwertzerlegung ([18]).

Bemerkung 1.2.9 (Bestapproximation mittels Rang-k-Matrizen) *Seien M und R wie in Satz 1.2.8 gegeben, wobei k für Σ_k bekannt ist. Dann gilt*

$$R = U\Sigma_k V^T = \begin{bmatrix} U' & U'' \end{bmatrix} \begin{bmatrix} \Sigma' & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V'^T \\ V''^T \end{bmatrix} \tag{1.23}$$

mit

$$\Sigma' := \Sigma_k(1:k, 1:k) \in \mathbb{R}^{k\times k}, \tag{1.24}$$

$$U' := U(1:n, 1:k) \in \mathbb{R}^{n\times k} \tag{1.25}$$

und

$$V' := V(1, m, 1:k) \in \mathbb{R}^{m\times k}. \tag{1.26}$$

Die Einträge von $U'' \in \mathbb{R}^{n\times n-k}$ und $V''\mathbb{R}^{m\times m-k}$ sind beliebig, denn das Matrixprodukt hängt nicht von U'' und V'' ab, da diese Blöcke der Matrix bei der Matrix-Matrix-Multiplikation mit 0 multipliziert werden. Dadurch können wir das Produkt als

$$R = U'\Sigma' V'^T \tag{1.27}$$

schreiben, und wir nennen (1.27) eine komprimierte Singulärwertzerlegung. Mit $A := U'\Sigma'$, $B := V'$ oder $A := U'$, $B := \Sigma' V'$ ist $R \in \mathcal{R}k(n, m)$.

Für die Approximation einer Rang-k-Matrix durch eine Rang-k'-Matrix im Fall $k' < k$ ist die allgemeine Singulärwertzerlegung zu teuer, weil die Singulärwertzerlegung die spezielle Produktstruktur der Rang-k-Matrizen ignoriert. Ein Algorithmus, der diese Struktur berücksichtigt, verwendet die QR-Zerlegung.

QR-Zerlegung

Definition 1.2.10 (QR-Zerlegung) *Sei $A \in \mathbb{R}^{n\times m}$ eine beliebige Matrix. Die Zerlegung*

$$A = QR \tag{1.28}$$

mit orthogonalem $Q \in \mathbb{R}^{n\times n}$ und oberer Dreiecksmatrix $R \in \mathbb{R}^{n\times m}$ heißt QR-Zerlegung der Matrix A.

Definition 1.2.11 (komprimierte QR-Zerlegung) *Seien Q und R die Faktoren der QR-Zerlegung der Matrix $M = QR \in \mathbb{R}^{n\times m}$, und es gelte $n > m$. In diesem Fall hat R die Blockstruktur $\begin{bmatrix} R' \\ 0 \end{bmatrix}$. R' ist eine $m \times m$- Matrix, die eine obere Dreiecksform hat. Betrachten wir nun die Teilmatrix $Q' := Q(1:n, 1:m)$, so gilt*

$$M = Q'R' \text{ mit } Q' \in \mathbb{R}^{n\times m} \text{ und } R' \in \mathbb{R}^{m\times m}. \tag{1.29}$$

Wir wollen (1.29) komprimierte QR-Zerlegung nennen.

Singulärwertzerlegung für Rang-k-Matrizen

Mittels der komprimierten QR-Zerlegung berechnen wir die komprimierte Singulärwertzerlegung einer Rang-k-Matrix.

Bemerkung 1.2.12 (Singulärwertzerlegung für Rang-k-Matrizen) *Die komprimierte Singulärwertzerlegung einer Rang-k-Matrix können wir mittels folgender Schritte berechnen.Sei $M := AB^T \in \mathcal{R}k(n,m)$ gegeben.*

1) *Berechne die komprimierte QR-Zerlegung von A. Man erhält $A = Q_A R_A$ mit $Q_A \in \mathbb{R}^{n\times k}$ und $R_A \in \mathbb{R}^{k\times k}$, wobei R_A eine obere Dreiecksmatrix ist.*

2) *Berechne die komprimierte QR-Zerlegung von B. Man erhält $B = Q_B R_B$ mit $Q_B \in \mathbb{R}^{m\times k}$ und $R_B \in \mathbb{R}^{k\times k}$, wobei R_B eine obere Dreiecksmatrix ist.*

3) *Berechne die Singulärwertzerlegung von $R_A R_B^T$. Man erhält $R_A R_B^T = \hat{U}\Sigma\hat{V}^T$ mit $\hat{U}, \Sigma, \hat{V}^T \in \mathbb{R}^{k\times k}$.*

4) *Definiere $U := Q_A\hat{U} \in \mathbb{R}^{n\times k}$ und $V := Q_B\hat{V} \in \mathbb{R}^{m\times k}$.*

Nun ist $M = U\Sigma V^T$ die komprimierte Singulärwertzerlegung von M. Die Aufwandsabschätzung dieses Algorithmus werden wir in Kapitel 2 durchführen und sehen, dass der Aufwand geringer ist als mit der Kürzung, die die Produktstruktur nicht berücksichtigt.

Bemerkung 1.2.13 (Kürzung einer Rang-k-Matrix) *Haben wir eine Matrix $R \in \mathcal{R}k(n,m)$, so können wir nun auf Grund von Bemerkung 1.2.12 eine Bestapproximation $\tilde{R} \in \mathcal{R}k'(n,m)$ mit $k' < k$ bezüglich der Spektral- oder Frobeniusnorm berechnen. Hierzu verwenden wir die Idee aus Satz 1.2.8. Haben wir die komprimierte Singulärwertzerlegung $R = U\Sigma V^T$ berechnet, so zerlegen wir Σ in die Blöcke $\begin{bmatrix} \Sigma' & 0 \\ 0 & \Sigma'' \end{bmatrix}$, $\Sigma' \in \mathbb{R}^{k'\times k'}$. Wir wählen nun entsprechend der Zerlegung von Σ eine Zerlegung für $U = \begin{bmatrix} U' & U'' \end{bmatrix}$ und $V = \begin{bmatrix} V'^T \\ V''^T \end{bmatrix}$. $\tilde{R}$ ergibt sich nun durch $\tilde{R} = U'\Sigma' V'$.*

Wollen wir eine gegebene Rang-k-Matrix $R \in \mathcal{R}k(n,m)$ kürzen, sodass die gekürzte Matrix den vorgegebenen Fehler $\epsilon \geq 0$ in der $\|\cdot\|_2$-Norm nicht überschreitet, so können wir dies mit folgenden Schritten gewährleisten:

1. *Berechne die komprimierte Singulärwertzerlegung von R. Wir erhalten*

$$R = U\Sigma V^T, \quad U \in \mathbb{R}^{n\times k},\ \Sigma \in \mathbb{R}^{k\times k},\ V \in \mathbb{R}^{m\times k},$$

   wobei $\sigma_1 \geq \ldots \geq \sigma_k \geq 0$ die Singulärwerte sind.

2. *Bestimme k_ϵ, sodass k_ϵ die größte Zahl mit*

$$\sigma_{k_\epsilon} \leq \epsilon \tag{1.30}$$

   ist.

3. Bestimme $\Sigma' \in \mathbb{R}^{k_\epsilon \times k_\epsilon}$ *und* $\tilde{R} := U'\Sigma'V'^T$ *wie in Bemerkung 1.2.9.*

Wir kennen nun einen Algorithmus zur Kürzung einer Rang-k-Matrix. Wir wissen aber noch nicht, wie man die QR-Zerlegung oder die Singulärwerte berechnen kann. Diese und andere technische Details der Algorithmen werden wir im zweiten Kapitel einführen, wo wir zwei Algorithmen für die Kürzung detailliert beschreiben und analysieren. Diese Algorithmen werden wir dann in Kapitel 3 parallelisieren.

1.3 Warum parallelisieren?

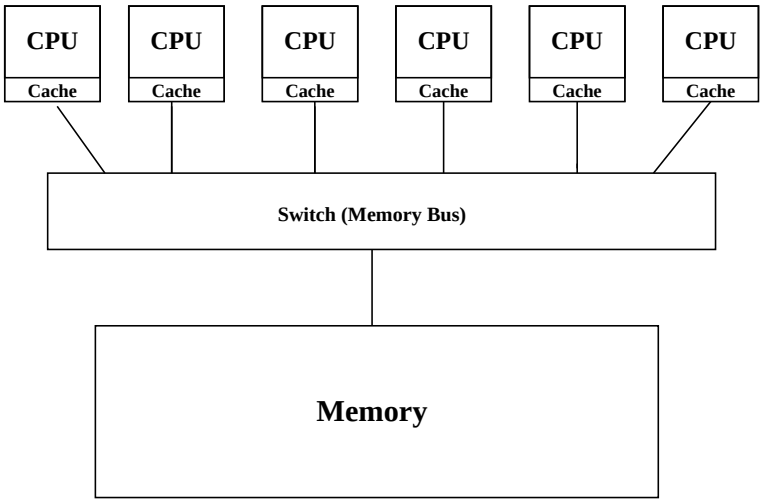
Die aktuell besten sequentiellen Algorithmen behandeln viele aktuelle Probleme mit linearer Komplexität oder fast linearer Komplexität. Deshalb stellt sich die Frage, warum man noch parallele Algorithmen benötigt.
Heutige Workstations mit einem Prozessorkern haben oft 4GByte Arbeitsspeicher. Wollen wir auf einem solchen Rechner eine quadratische vollbesetzte Matrix mit doppelter Genauigkeit im Arbeitsspeicher speichern, dann darf diese maximal 23170 Zeilen und Spalten haben, denn jeder Eintrag benötigt 8Byte ([26]). Bei 8GByte Arbeitsspeicher erhalten wir für eine quadratische Matrix eine maximale Größe von 32768×32768. Hier sehen wir auch das quadratische Verhalten des Speicherbedarfs von vollbesetzten Matrizen. In der Praxis können bei Integralgleichungen Gleichungssysteme mit einer Million oder mehr Unbekannten auftreten. Diese Gleichungssysteme können wir Dank Hierarchischer Matrizen auf Workstations bearbeiten, weil der Speicheraufwand dann nur $\mathcal{O}(n \log n)$ beträgt.
Größere Matrizen lassen sich nur noch auf Rechnern mit größerem Arbeitsspeicher verarbeiten. Dies können Shared-Memory-Systeme oder Distributed-Memory-Systeme sein. Solche Systeme zeichnen sich dadurch aus, dass sie mehrere Prozessoren haben.
Ein weiteres Problem ist die Laufzeit der Algorithmen, weil diese sich oftmals proportional zur Speicherkomplexität verhält. Dieses Problem können wir teilweise umgehen, indem wir für große Probleme mehrere Prozessoren zu Berechnung verwenden.

1.3.1 Verschiedene parallele Computersysteme

Wir werden die Algorithmen auf Systemen mit geteiltem („shared memory“) und verteiltem („distributed memory“) Arbeitsspeicher testen. Wir geben hier eine kurze Erklärung beider Systeme. Genauere Informationen findet man in [4, 29, 30].

Shared-Memory-Systeme

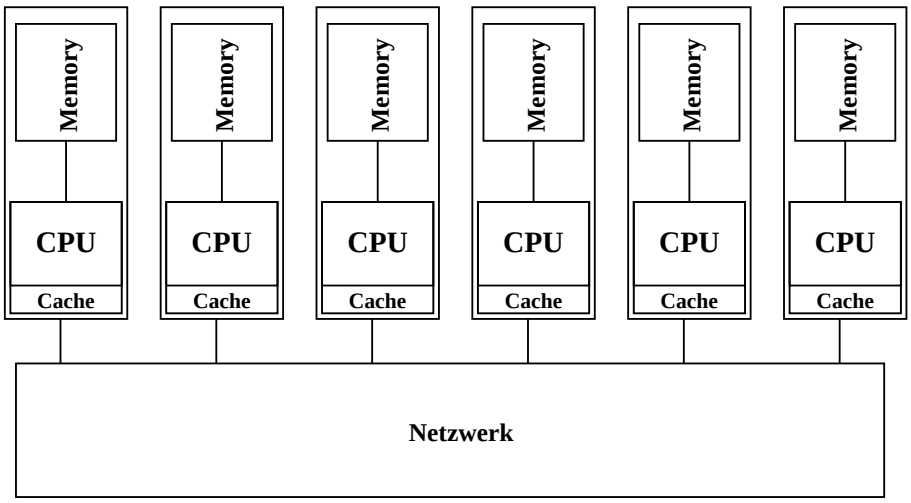
Bei einem Shared-Memory-System haben alle Prozessoren Zugriff auf den kompletten Arbeitsspeicher des Systems. Bei heutigen Systemen haben die Prozessoren aber häufig noch einen Cache-Speicher, auf den nur ein Prozessor lokal zugreifen kann.



Dabei entstehende Probleme, zum Beispiel dass zwei Prozessoren die gleichen Daten im Cache-Speicher halten, verändern oder zurückschreiben wollen, werden von der Hardware abgefangen.
Ein Nachteil dieser Systeme ist, dass der Memory Bus ein Flaschenhals für den Speicherzugriff sein kann.

Distributed-Memory-Systeme

Bei Distributed-Memory-Systemen hat jeder Prozessor einen lokalen Arbeitsspeicher. Die Prozessoren sind durch ein Netzwerk verbunden.



Bei diesen Systemen müssen wir bei den meisten parallelen Algorithmen Daten über das Netzwerk austauschen, wobei dieser Nachrichtenaustausch um einiges teurer ist als ein direkter Zugriff auf den Arbeitsspeicher.
Ein Vorteil dieser Distributed-Memory-Systeme ist, dass sie praktisch aus vielen

Einzelrechnern zusammengebaut werden können. Dadurch sinken die Anschaffungskosten für solche Systeme.

Kapitel 2

Sequentielle Kürzung

In Kapitel 1 haben wir die Rang-k-Matrizen definiert und die Kürzung mittels der Singulärwerte kennengelernt. Nun wollen wir die einzelnen Schritte der Kürzung exakt anhand eines Algorithmus in Pseudocode beschreiben.
Unser Kürzungsalgorithmus besteht im Wesentlichen aus Matrix-Multiplikationen, QR-Zerlegungen und der Berechnung der Singulärwerte. Wir werden in diesem Kapitel zwei verschiedene Algorithmen für die Kürzung vorstellen. Beide Algorithmen benötigen die QR-Zerlegung, die wir mittels Householder-Reflexionen berechnen. Die Singulärwerte werden unterschiedlich berechnet.
Im ersten Kürzungsalgorithmus berechnen wir die Singulärwerte mittels Singulärwertzerlegung. Der zweite Kürzungsalgorithmus berechnet die Singulärwerte unter Verwendung der Lösung eines Eigenwertproblems.
Wie wir später sehen werden, werden die Singulärwertzerlegung und die Lösung des Eigenwertproblems immer für Matrizen der Größe $k \times k$ gerechnet. Anwendungen der Rang-k-Matrizen gewährleisten in der Regel, dass $k \ll n, m$ gilt. Die Technik der Hierarchischen Matrizen erreicht dies zum Beispiel durch die passende Wahl einer Zulässigkeitsbedingung und einer Fehlerschranke (z.B. [18]). Wir nehmen nun an, dass $k \ll n, m$ in den folgenden Teilen der Arbeit immer gilt.
Durch $k \ll n, m$ sind die Matrizen für das Eigenwertproblem und die Singulärwertzerlegung klein. Daher werden wir den Lösungsalgorithmus des Eigenwertproblems und die Singulärwertzerlegung nicht parallelisieren. Wir werden für diese beiden Teilprobleme existierende sequentielle Algorithmen verwenden.

2.1 Algorithmen

Die Algorithmen werden mittels Pseudocode angegeben. Wir werden am Anfang eines Algorithmus definieren, was die Übergabetypen sind und was der Algorithmus berechnet. Die erste Zeile des Algorithmus dient zur Deklaration der benötigten Variablen. Die Variablen, die wir für die Schleifen benötigen, werden wir nicht extra deklarieren.
Wir werden zuerst den QR-Algorithmus einführen, den wir für die beiden Kürzungsalgorithmen benötigen.

2.1.1 Die QR-Zerlegung

Wir wollen die Matrix $A \in \mathbb{R}^{n\times m}, n > m$, in die Faktoren Q und R zerlegen, sodass $A = QR$ gilt. Dabei benötigen wir nur die komprimierte QR-Zerlegung, wie sie in (1.29) beschrieben ist. Dadurch ist Q keine orthogonale Matrix. Die Spalten der Matrix sind aber weiterhin orthonormal zueinander, und es gilt

$$Q \in \mathbb{R}^{n\times m} \text{ und } R \in \mathbb{R}^{m\times m}. \tag{2.1}$$

Der Algorithmus wendet dabei die sogenannten Householder-Reflexionen auf die Matrix A an und erstellt so die Matrix R der QR-Zerlegung. Eine Householder-Reflexion P ist eine Matrix und hat folgende Form ([17]):

$$P = I - \frac{2}{v^T v} v v^T \quad \in \mathbb{R}^{n\times n}, \quad v \in \mathbb{R}^n. \tag{2.2}$$

Der Vektor v wird Householdervektor genannt.
Der Algorithmus 1 berechnet für einen Vektor x den Householdervektor so, dass Px ein Vielfaches des Einheitsvektors $e_1 := (1, 0 \ldots, 0)$ ist ([17]). Von nun an wollen wir $\beta := \frac{2}{v^T v}$ als Skalierungsfaktor verwenden.

Algorithmus 1 householdervector(v, n, x)

Benötigt: Die Funktion benötigt den Speicher für den Householdervektor $v \in \mathbb{R}^n$ und dessen Größe n. x repräsentiert eine Teilspalte aus der Matrix A der Länge n.

Beschreibung: In v wird der Householdervektor abgespeichert und β wird zurückgegeben.

1: $\sigma, x_0, \mu \in \mathbb{R}$
2: $\sigma := x(2:n)^T x(2:n)$;
3: **if** $\sigma = 0$ **then**
4: $\quad \beta := 0$;
5: **else**
6: $\quad x_0 := x(1)$;
7: $\quad \mu := \sqrt{x_0 \cdot x_0 + \sigma}$;
8: $\quad$ **if** $x_0 \leq 0.0$ **then**
9: $\quad\quad v(1) := x_0 - \mu$;
10: $\quad$ **else**
11: $\quad\quad v(1) := -\frac{\sigma}{x_0 - \mu}$;
12: $\quad$ **end if**
13: $\quad \beta := \frac{2v(1)^2}{\sigma + v(1)^2}$;
14: **end if**
15: $v(2:n) := \frac{x(2:n)}{v(1)}$;
16: $v(1) := 1$;
17: **return** β;

Nach jedem Algorithmus geben wir eine kurze Erklärung zum besseren Verständnis.

Bemerkung 2.1.1 (Erklärung des Algorithmus 1) *Dem Algorithmus wird der Vektor $x \in \mathbb{R}^n$ übergeben, und er berechnet den Householdervektor $v \in \mathbb{R}^n$. Die Rechenschritte findet man in [17].*

- *Zeile 1 dient einfach zur Allokation bzw. Definition verschiedener Variablen. Wir werden diese Zeile in jedem Algorithmus wiederfinden aber bei der Erklärung nicht mehr erwähnen.*
- *Zeile 2 berechnet ein Skalarprodukt des Vektors $x(2:n)$.*
- *Die Zeilen 3 bis 14 dienen zur Berechnung von β und eines Skalierungsfaktors, der in $v(1)$ gespeichert wird.*
- *In Zeile 15 und Zeile 16 wird der Householdervektor definiert. Es ist zu beachten, dass $v(1)$ immer 1 ist.*
- *Zeile 17 gibt β an die aufrufende Prozedur zurück.*

Übergeben wir dem Algorithmus den Vektor $A(:,1)$, so erhalten wir den Householdervektor v_1, durch den $P_1 := I - \beta_1 v_1 v_1^T$ definiert ist. Die 1 gibt an, dass es sich um die Reflexion und den Vektor für die erste Spalte handelt. Wenden wir diese Projektionsmatrix auf $A(:,1)$ an, so gilt nach [17, Kapitel 5.1]

$$P_1 A(:,1) = (I - \beta_1 v v^T) A(:,1) = (c, 0, \ldots, 0)^T, \quad c > 0. \tag{2.3}$$

Der Algorithmus wendet die Householderreflexion auf die komplette Matrix an und berechnet daraufhin den Householdervektor für die nächste Spalte der Matrix. Dabei ist zu beachten, dass der zweite Householdervektor nur für den Teil $(P_1 A)(2:n,2) \in \mathbb{R}^{n-1}$ der zweiten Spalte berechnet wird. Wir erhalten nun den Householdervektor $v_2 \in \mathbb{R}^{n-1}$. Mittels

$$\tilde{P}_2 = (I - \beta_2 v_2 v_2^T) \in \mathbb{R}^{n-1 \times n-1} \tag{2.4}$$

können wir die Reflexion P_2 durch

$$P_2 := \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P} \end{bmatrix} \tag{2.5}$$

definieren. Die i-te Reflexion ist somit durch

$$P_i := \begin{bmatrix} I_{i-1} & 0 \\ 0 & \tilde{P}_i \end{bmatrix} \quad \forall i \in \{1, \ldots, n\} \tag{2.6}$$

definiert. Dabei ist $I_{i-1} \in \mathbb{R}^{i-1 \times i-1}$ die Einheitsmatrix und $\tilde{P}_i := (I_{n-i+1} - \beta_i v_i v_i^T)$ die Projektionsmatrix zu dem Householdervektor v_i. Dieser Householdervektor hängt vom Vektor $(P_1 \cdot \ldots \cdot P_{i-1} A)(i:n,i)$ ab. Die Matrix Q ergibt sich nun aus dem Produkt der P_i, $i \in \{1, \ldots, n\}$, Matrizen mittels

$$\tilde{Q} = P_n \cdot P_{n-1} \cdot \ldots \cdot P_1 \in \mathbb{R}^{n \times n}, \tag{2.7}$$

durch die Einschränkung

$$Q := \tilde{Q}(1:n, 1:m). \tag{2.8}$$

R ergibt sich mittels

$$\tilde{R} := \tilde{P}A \tag{2.9}$$

und

$$R := \tilde{R}(1:m, 1:m). \tag{2.10}$$

Diese Schritte berechnet der Algorithmus 2. In der ersten Schleife berechnen wir die Matrix R und speichern die β_i und die Vektoren v_i in der Matrix A. Dies ist möglich, weil bei der Anwendung der i-ten Reflexion, die ersten $i-1$ Spalten unverändert bleiben. In der zweiten Schleife wird die Matrix Q berechnet und in der Matrix A gespeichert.
Der Algorithmus verwendet dabei weitere Optimierungen zur Reduktion des Rechenaufwands. Wie oben beschrieben überschreiben wir die Matrix A mit der Matrix Q. Außerdem wissen wir aufgrund der speziellen Struktur der Matrizen P_i, dass bei der Multiplikation mit einer Matrix $X \in \mathbb{R}^{n\times m}$ nur der Teil $X(i : n : i : m)$ verändert wird. Dadurch können wir den i-ten Schritt, $i \in \{1, \dots m\}$,

$$P_i(P_{i-1} \cdot \ldots \cdot P_1 A) \tag{2.11}$$

durch

$$\tilde{P}_i((P_{i-1} \cdot \ldots \cdot P_1 A)(i:n, i:m)) \tag{2.12}$$

ersetzen. Das Ergebnis soll wiederum in A gespeichert werden, wodurch wir für den i-ten Schritt

$$A(i:n, i:m) := A(i:n, i:m) - \beta_i v_i v_i^T A(i:n, i:m) \tag{2.13}$$

erhalten. Wir werden den Teil

$$A(i:n, i:m) - \beta_i v_i v_i^T A(i:n, i:m) \tag{2.14}$$

im Algorithmus so umformulieren, dass wir die Operation mit Skalarprodukten und Vektoradditionen berechnen können.
Für den i-ten Schritt berechnen wir zuerst die skalierten Skalarprodukte:

$$s_j := \beta_i v_i^T A(i:n, j), \quad j \in \{i, \dots, m\}. \tag{2.15}$$

Die Skalare s_j verwenden wir dann bei der Vektoraddition

$$A(i:n, j) - s_j v_i, \quad j \in \{i, \dots, m\}. \tag{2.16}$$

Dies sind alle technischen Details, die wir für den Algorithmus benötigen.

Algorithmus 2 qr_decomposition(A,R)

Benötigt: $A \in \mathbb{R}^{n\times k}, R \in \mathbb{R}^{k\times k}$

Beschreibung: Berechnet die QR-Zerlegung von A, wobei Q in A gespeichert wird.

1: $\beta \in \mathbb{R},\ s \in \mathbb{R}^k,\ v \in \mathbb{R}^n$;
2: **for** $i = 1, \ldots, k$ **do**
3: $\beta :=$ householdervector$(v(i:n),\ n-i+1,\ A(i:n,i))$;
4: **for** $j = i, \ldots, k$ **do**
5: $s(j) := \beta v(i:n)^T A(i:n,j)$;
6: **end for**
7: **for** $j = i, \ldots, k$ **do**
8: $A(i:n,j) := A(i:n,j) + s(j)v(i:n)$;
9: **end for**
10: $R(i, i:k) := A(i, i:k)$;
11: $A(i, i:k) := 0.0$;
12: $A(i+1:n, i) := v(i+1:n)$;
13: $A(i,i) := \beta$;
14: **end for**
15: **for** $i = k, \ldots, 1$ **do**
16: $\beta := A(i,i)$;
17: $v(i) := 1$;
18: $v(i+1, n) := A(i+1:n, i)$;
19: $A(i,i) := 1$;
20: $A(i+1:n, i) := 0$;
21: **for** $j = i, \ldots, k$ **do**
22: $s(j) := \beta v^T A(i:n,j)$;
23: **end for**
24: **for** $j = i, \ldots, k$ **do**
25: $A(i:n,j) := A(i:n,j) - s(j)v(i:n)$;
26: **end for**
27: **end for**

Bemerkung 2.1.2 (Erklärung des Algorithmus 2)

- *Die Zeilen 2 bis 14 bilden eine Schleife, die k-mal durchlaufen wird. Nach dieser Schleife ist R berechnet*
  - *Die Zeile 3 ruft die Prozedur zur Berechnung des Householdervektors auf. Danach sind v und β bekannt.*
  - *Die Zeilen 4 bis 6 berechnen die Skalare aus (2.15).*
  - *Die Zeilen 7 bis 9 berechnen (2.16). Danach enthält im i-ten Schritt $A(i, i:k)$ die Einträge der i-ten Zeile der oberen Dreiecksmatrix R.*
  - *Die Zeilen 10 und 11 speichern im i-ten Schritt die i-te Zeile der Matrix R und setzen die entsprechenden Einträge in A auf* 0.

  - *In Zeile 12 wird der Householdervektor in der Matrix A gespeichert. $v(1)$ muss hier nicht gespeichert werden, weil $v(1) = 1$ gilt für alle Householdervektoren, die wir mit Algorithmus 1 berechnen.*
  - *Zeile 13 speichert β. Die Spalte, in der v und β gespeichert werden, wird durch (2.13) in den nachfolgenden Schritten der Schleife nicht mehr verändert.*

- *Die Zeilen 15 bis 27 bilden eine Schleife, die k-mal durchlaufen wird. Diese Schleife berechnet die Matrix Q und läuft in umgekehrter Reihenfolge.*
  - *Die Zeilen 16 bis 20 lesen den Householdervektor v und β aus der Matrix A aus. Zusätzlich wird in die Spalte in der diese Daten standen der Einheitsvektor geschrieben, damit die Multiplikation (2.8) durchgeführt werden kann.*
  - *Zeilen 21 bis 26 berechnen mittels (2.15) und (2.16) die Produkte der Reflexionsmatrizen zur Berechnung von Q (siehe (2.8)).*

2.1.2 Kürzung mittels Singulärwertzerlegung

Die Kürzung anhand Singulärwertzerlegung funktioniert nun wie in Kapitel 1 beschrieben. Der Kürzungsalgorithmus berechnet die Bestapproximation für die Spektralnorm. Die Singulärwertzerlegung wird von der Funktion

$$\mathrm{svd}(A, U, S, V) \tag{2.17}$$

berechnet. Eine mögliche Implementation des Algorithmus für die Singulärwertzerlegung finden wir in [17]. Wir werden die Singulärwertzerlegung nur für quadratische Matrizen berechnen. Daher gilt

$$A, U, S, V \in \mathbb{R}^{k \times k}. \tag{2.18}$$

Das Ergebnis der Singulärwertzerlegung wird in die Matrizen U, S, V geschrieben. Am Ende werden die Matrizen A und B der Rang-k-Matrix aktualisiert. Hier muss man beachten, dass der Rang kleiner wird und Speicher freigegeben werden kann.
Unser erster Algorithmus, der eine Rang-k-Matrix Kürzung berechnet, ist Algorithmus 3.

Algorithmus 3 truncation_svd(A,B,ϵ)

Benötigt: $A \in \mathbb{R}^{n\times k}, B \in \mathbb{R}^{m\times k}, \epsilon$.

Beschreibung: $A \in \mathbb{R}^{n\times k_{new}}, B \in \mathbb{R}^{m\times k_{new}}$ sodass der maximale Fehler ϵ in der Spektralnorm mit minimalem k_{new} erfüllt ist.

1: $k_{new} \in \mathbb{R}, R, R_A, R_B, U, V, S \in \mathbb{R}^{k\times k}$;
2: qr_decomposition(A,R_A);
3: qr_decomposition(B,R_B);
4: $R := R_A R_B^T$;
5: svd(R,U,S,V);
6: $k_{new} := k$;
7: **for** $i = k, \ldots, 1$ **do**
8: **if** $S(i,i) < \epsilon$ **then**
9: $k_{new} := i$;
10: **end if**
11: **end for**
12: $A := A(1:n, 1:k_{new})U(1:k_{new}, 1:k)$;
13: $B := B(1:m, 1:k_{new})V(1:k_{new}, 1:k)S(1:k_{new}, 1:k_{new})^T$;

Bemerkung 2.1.3 (Erklärung des Algorithmus 3) *Dieser Algorithmus berechnet nun die QR-Zerlegung nach Bemerkung 1.2.13 mit dem kleinstmöglichen Fehler in der Spektralnorm.*

- *Die Zeilen 4 bis 11 dienen zur Berechnung der Singulärwertzerlegung und der Bestimmung des neuen Ranges k_{new} der gekürzten Rang-k-Matrix.*
- *Die Zeilen 12 und 13 definieren die gekürzten Faktoren A und B der nun gekürzten Rang-k_{new}-Matrix.*

2.1.3 Kürzung mittels Eigenwertiteration

Die Kürzung mittels einer Orthogonalen Iteration wurde in [18] als Alternative zur Methode mit der exakten Singulärwertzerlegung erwähnt, aber aufgrund hoher Kosten verworfen. Wir werden hier eine modifizierte Version dieses Algorithmus analysieren und mit der Kürzung mittels Singulärwertzerlegung vergleichen. Der Algorithmus berechnet die Singulärwertzerlegung nicht direkt wie die obige Methode, sondern wir gehen nun den Umweg über ein Eigenwertproblem. Dazu wollen wir die Linkssingulärvektoren und die Rechtssingulärvektoren definieren.

Definition 2.1.4 (Links- und Rechtssingulärvektoren [33]) *Sei $M \in \mathbb{R}^{n\times m}$ und sei $M = U\Sigma V^T$ die Singulärwertzerlegung nach Satz 1.2.6. Die Matrix U besteht aus den Vektoren*

$$U(:,1), \ldots, U(:,n) \in \mathbb{R}^n, \tag{2.19}$$

diese nennt man Linkssingulärvektoren von A. Die Vektoren

$$V(:,1), \ldots, V(:,m) \in \mathbb{R}^m \tag{2.20}$$

nennt man Rechtssingulärvektoren.

Bemerkung 2.1.5 ([33]) *Sei $M \in \mathbb{R}^{n\times m}$ und sei $M = U\Sigma V^T$ die Singulärwertzerlegung. Die Linkssingulärvektoren $U(:,1),\ldots,U(:,n)$ sind die orthonormierten Eigenvektoren von MM^T, die Rechtssingulärvektoren $V(:,1),\ldots,V(;,m)$ sind die orthonormierten Eigenvektoren von M^TM.*

Betrachten wir nun eine Rang-k-Matrix $AB^T \in \mathbb{R}^{n\times m}$. Dann enthält nach obiger Definitionen die Matrix V^T der Singulärwertzerlegung $AB^T = U\Sigma V^T$ die Eigenvektoren der Matrix

$$(AB^T)^T AB^T = BA^T AB^T. \tag{2.21}$$

Sei nun das Eigenwertproblem

$$BA^T AB^T y = \lambda y \tag{2.22}$$

gegeben. Dabei ist $y \in \mathbb{R}^m$ ein Eigenvektor und λ der zugehörige Eigenwert. Wir wollen das Eigenwertproblem nun so lösen, dass wir alle Eigenwerte und Eigenvektoren bestimmen. Die Eigenvektoren wollen wir in Zukunft in der Matrix $Y \in \mathbb{R}^{m\times m}$ speichern und die Eigenwerte in der gleichen Anordnung in der Matrix $\Lambda \in \mathbb{R}^{m\times m}$. Wir lösen aber das durch die Matrix B transformierte Eigenwertproblem. Setzen wir $y = Bx, x \in \mathbb{R}^k$ so erhalten wir das kleinere Eigenwertproblem

$$A^T AB^T Bx = \lambda x. \tag{2.23}$$

Lösen wir (2.23) und speichern die Eigenvektoren in der Matrix $X \in \mathbb{R}^{k\times k}$, so erhalten durch $BX = Y$ die Eigenvektoren des Eigenwertproblems (2.22). Damit in Y die Rechtssingulärvektoren enthalten sind, müssen wir Y orthogonalisieren. Mittels einer QR-Zerlegung erhalten wir $Y = QR$, und setzen wir $Y' = Q$, so enthält Y' die Rechtssingulärvektoren. Die Anordnung ist identisch mit der Anordnung von Λ.
Betrachten wir nun die Singulärwertzerlegung $AB^T = U\Sigma V^T$. Es gilt

$$(AB^T)^T AB^T = (U\Sigma V^T)U\Sigma V^T = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T. \tag{2.24}$$

Wir wissen zusätzlich, dass für eine Matrix $M \in \mathbb{R}^{n\times m}$

$$\sigma_i = \sqrt{\lambda_i} \quad \forall \lambda_i \in \sigma(M^T M) \tag{2.25}$$

gilt, wobei $\sigma(M^TM)$ das Spektrum der Matrix M^TM ist, die σ_i sind die Singulärwerte von M.
Daher gilt nach (2.25), dass die Matrix Λ auf der Diagonalen die Quadrate der Singulärwerte von AB^T enthält. Wir müssen hier nur beachten, das die Einträge nicht so geordnet sind, wie es die Singulärwertzerlegung verlangt. Somit haben wir V und Σ der Singulärwertzerlegung von $(AB^T)^T AB^T$ berechnet.
Berechnen wir nun $AB^TY' = Z$, so ist Z bis auf die Anordnung identisch mit $U\Sigma$. Somit enthält Z die mit den Singulärwerten skalierten Linkssingulärvektoren. Die Singulärwerte erhalten wir nun, indem wir die Normen $\|Z(:,i)\|_2$ berechnen. Um diese Singulärwerte zu erhalten, haben wir nur die Matrix X verwendet. Die Eigenwerte benötigen wir nicht und werden diese im Algorithmus auch nicht berechnen.

Wir wollen nun die Bestapproximation zu einem $\epsilon > 0$ für die Spektralnorm wie in Kapitel 1 bestimmen. Seien nun $i_1, \ldots, i_{k'}$ alle Indizes, für die $\|Z(:, i_j)\|_2 \geq \epsilon, \forall j \in \{1, \ldots, k^{'}\}$ gilt, dann erfüllt die Rang-k-Matrix

$$A^{'}(B^{'})^T \quad \text{mit } A^{'} := [Z(:, i_1), \ldots Z(:, i_{k'})], B^{'} := [Y^{'}(:, i_1), \ldots Y^{'}(:, i_{k'})] \quad (2.26)$$

die Bedingungen der Bestapproximation.
Der Algorithmus hat aber die Schwäche, dass die quadrierten Singulärwerte über das Eigenwertproblem bestimmt werden. Dadurch erhalten wir nur eine einfache Genauigkeit für die Singulärwerte der Rang-k-Matrix. Dies trifft somit auch auf die Eigenvektoren in der Matrix V zu. Um die Genauigkeit zu erhöhen, können wir die Orthogonale Iteration aus [17, Chapter 7.3.2] verwenden. Die Konvergenzrate dieser Iteration hängt dabei von den Eigenwerten der Rang-k-Matrix ab. Für die Anwendung der Hierarchischen Matrizen erhalten wir oftmals stark fallende Singulärwerte für die Rang-k-Matrizen (siehe [18, Kapitel 8]). Deshalb werden wir nur zwei Iterationen der Orthogonalen Iteration durchführen, um eine doppelte Genauigkeit zu erreichen.
Die Lösung des Eigenwertproblems und somit die Berechnung von X wird in dem folgenden Algorithmus von der Funktion

$$\text{evp}(A, X), \quad (2.27)$$

$A, X \in \mathbb{R}^{k \times k}$, übernommen. Es bietet sich an, für diese Methode LAPACK-Routinen zu verwenden. Die Methode berechnet dabei die Eigenvektoren der Matrix A und schreibt diese in die Matrix X.
Für die Auswahl der Spalten wollen wir unsere Spaltennotation noch etwas erweitern.

Notation 2.1.6 *Sei $A \in \mathbb{R}^{n \times m}$ eine Matrix und $I := \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, m\}$ eine Teilmenge der Spaltenindizes, dann ist die Teilmatrix $A(\sigma : \tau, I)$ definiert durch*

$$A(\sigma : \tau, I) := [A(\sigma : \tau, i_1)\ A(\sigma : \tau, i_2)\ \ldots\ A(\sigma : \tau, i_k)] \in \mathbb{R}^{\tau - \sigma \times k}. \quad (2.28)$$

Wir werden diese Notation in unserem zweiten Kürzungsalgorithmus benötigen. Dieser ist nun folgendermaßen definiert:

Algorithmus 4 truncation_evp(A,B,ϵ)

Benötigt: $A \in \mathbb{R}^{n\times k}, B \in \mathbb{R}^{m\times k}, \epsilon$

Beschreibung: Berechnet $A \in \mathbb{R}^{n\times k'}, B \in \mathbb{R}^{m\times k'}$, sodass der maximale Fehler ϵ mit minimalem k' eingehalten wird.

1: $G, G_A, G_B, X, R \in \mathbb{R}^{k\times k}$, $\tilde{B} \in \mathbb{R}^{m\times k}$, $\tilde{A} \in \mathbb{R}^{n\times k}$;
2: $G_A = A^T A$;
3: $G_B = B^T B$;
4: $G = G_A G_B$;
5: evp(G, X);
6: $\tilde{B} = BX$;
7: **for** $i = 1, 2$ **do**
8: $\quad G = B^T \tilde{B}$;
9: $\quad G_B = G_A G$;
10: $\quad \tilde{B} = BG_B$;
11: $\quad$ qr_decomposition($\tilde{B}$,R);
12: **end for**
13: $G = B^T \tilde{B}$;
14: $\tilde{A} = AG$;
15: $I = \{i \in \{1, \ldots k\} \mid \|\tilde{A}(\cdot, i)\|_2 > \epsilon\}$;
16: $A := \tilde{A}(i, I)$;
17: $B := \tilde{B}(i, I)$;

Bemerkung 2.1.7 (Erklärung des Algorithmus 4) *Der Algorithmus 4 berechnet die Lösung mittels der Lösung eines Eigenwertproblems und einer Orthogonalen Iteration.*

- *Die Zeilen 2 bis 6 berechnen die Eigenvektoren von (2.22).*
- *Die Zeilen 7 bis 12 enthalten die Orthogonale Iteration aus [17].*
- *Die Zeilen 13 bis 17 dienen zur Bestimmung der Singulärwerte und der Ermittlung der Spalten, die für die Faktoren A und B der gekürzten Rang-k-Matrix nach der Kürzung noch benötigt werden. Die gekürzte Matrix hat darauf den Rang $\#I$.*

2.2 Laufzeitanalyse

In diesem Kapitel werden wir die Komplexität der verschiedenen Algorithmen untersuchen. Wir werden nur Rechenoperationen zählen, weil wir annehmen, dass Speicheroperationen kostenlos sind. Diese Komplexitätsanalyse ist nur ein theoretisches Mittel, um das Laufzeitverhalten der Algorithmen auf einem Rechner abzuschätzen. Die Laufzeit auf einer normalen Workstation oder später auch auf parallelen Rechensystemen wird zusätzlich noch von der Größe des Arbeitsspeichers und des Caches beeinflusst, sowie von der Übertragungsgeschwindigkeit der Daten. Weitere Einflussfaktoren sind Pipelining und Scheduling [7].

Bemerkung 2.2.1 *In der folgenden Tabelle finden wir die Komplexitätsangabe für einfache algebraische Operationen, die wir in unseren Algorithmen verwenden.*

Operation	*Parameter*	*Komplexität*
$\alpha \cdot v,\ v + w$	$\alpha \in \mathbb{R}, v, w \in \mathbb{R}^n$	$C_s n$
$w^T \cdot v$	$v, w \in \mathbb{R}^n$	$C_{sp} n$
$\|v\|_2$	$v \in \mathbb{R}^n$	$C_{no} n$
$\alpha(w^T \cdot v)$	$\alpha \in \mathbb{R}, v, w \in \mathbb{R}^n$	$C_{ssp} n$
$A \cdot B$	$A \in \mathbb{R}^{n\times k}, B \in \mathbb{R}^{k\times m}$	$C_{mm} nmk$

Wie bei (1.12) angegeben werden wir die Komplexität der Algorithmen mit

$$\mathcal{N}_{Alg}(\cdot).$$

bezeichnen.

2.2.1 QR-Algorithmus

Für die Komplexitätsuntersuchung des QR-Algorithmus benötigen wir zuerst die Komplexität von Algorithmus 1, der den Householdervektor berechnet.

Lemma 2.2.2 *Die Komplexität der Berechnung eines Householdervektors der Länge n lässt sich durch*

$$\mathcal{N}_{house}(n) \leq C_{house} n \tag{2.29}$$

abschätzen.

Beweis: Die Zeilen 7, 9, 11 und 13 sind von n unabhängig und benötigen jeweils eine konstante Anzahl an Rechenschritten. Wie fassen diesen Aufwand mit der Konstanten C_1 zusammen. In Zeile 2 wird ein Skalarprodukt berechnet und in Zeile 15 wird ein Vektor der Länge n skaliert. Wir erhalten einen Gesamtaufwand von

$$C_1 + (C_s + C_{sp})n \leq C_{house} n, \quad \text{mit } C_1 + C_s + C_{sp} =: C_{house}. \tag{2.30}$$

■

Lemma 2.2.3 (Aufwand QR-Zerlegung) *Die QR-Zerlegung einer Matrix $A \in \mathbb{R}^{n\times k}$ mittels Algorithmus 2 hat einen Aufwand von*

$$\mathcal{N}_{QR}(n, k) = C_{QR} k^2 n \tag{2.31}$$

Beweis: Der Algorithmus 2 besteht aus zwei Hauptschleifen (Zeile 2 und 15), der Länge k. In der ersten Schleife haben wir folgenden Aufwand:

- Zeile 3 benötigt nach Lemma 2.2.2 im i-ten Schleifendurchlauf

  $$C_{house}(n - i + 1)$$

  Rechenschritte.

- Die innere Schleife von Zeile 4 bis 6 benötigt maximal k Durchläufe (Zeile 2), wobei jeder Durchlauf ein Skalarprodukt mit Vektoren der maximalen Länge n berechnet. Daher schätzen wir den Aufwand hierfür mit $C_{sp}nk$ ab.
- Die innere Schleife von Zeile 7 bis 9 benötigt wieder maximal k Durchläufe (Zeile 2), wobei jeder Durchlauf maximal C_skn Rechenschritte kostet (Multiplikation eines Skalars mit einem Vektor).
- Die restlichen Zeilen der ersten Schleife sind Speicheroperationen und sind nach unserer Annahme kostenlos.

In der Summe erhalten wir

$$k(C_{house}(n-i+1)+C_{sp}nk+C_skn) \overset{C_{house}+C_{sp}+C_s=:C_3}{\leq} k(C_3nk) = C_3k^2n \quad (2.32)$$

Für die zweite Schleife (Zeile 15 bis 27) können wir den Aufwand durch den Aufwand der ersten Schleife abschätzen, weil die Schleifen von Zeile 21 bis 23 und von Zeile 24 bis 26 auch in der ersten Schleife vorkommen. Ansonsten werden in der zweiten Schleife nur Werte kopiert. Somit erhalten wir mit

$$C_{QR} := 2C_3 \quad (2.33)$$

eine obere Schranke für den Aufwand:

$$\mathcal{N}_{QR}(n,k) \leq C_{QR}k^2n. \quad (2.34)$$

■

2.2.2 Kürzung mittels Singulärwertzerlegung

Kommen wir nun zu der Komplexitätsabschätzung unseres ersten Kürzungsalgorithmus. Wie bei der Definition des Algorithmus erwähnt verwenden wir für die Singulärwertzerlegung eine externe Funktion (z.B. aus [27]).

Bemerkung 2.2.4 (Komplexität Singulärwertzerlegung) *In [17, Kapitel 5.4] wird die Komplexität zu Berechnung der Faktoren U, S und V für eine Matrix $M \in \mathbb{R}^{n\times n}$ mit $21n^3$ angegeben. Wir geben den Aufwand mit*

$$\mathcal{N}_{svd}(n) \leq C_{svd}n^3 \quad (2.35)$$

an.

Lemma 2.2.5 (Aufwand der Kürzung mittels SVD) *Die Kürzung einer Rang-k-Matrix $R = AB^T \in \mathbb{R}^{n\times m}, n > m$ mittels Algorithmus 3 hat eine Komplexität von*

$$\mathcal{N}_{trsvd}(n,m,k) \leq C_{trsvd}k^2n. \quad (2.36)$$

Beweis:

- Die Zeilen 2 und 3 beschreiben jeweils eine QR-Zerlegung, wobei wir den Aufwand durch $n > m$ mit $2C_{QR}k^2n$ abschätzen können.
- Zeile 4 beschreibt eine Matrixmultiplikation mit einem Aufwand von $C_{mm}k^3$
- Zeile 5 kostet nach Bemerkung 2.2.4 $C_{svd}k^3$.
- Die Schleife bei Zeile 7 besteht nur aus Vergleichen und Zuweisungen und verursacht daher keine Kosten.
- Zeile 12 kostet $C_{mm}nk^2$
- In Zeile 13 kostet die Multiplikation von VS, wenn wir ausnutzen, dass S eine Diagonalmatrix ist, maximal C_1k^2, $C_1 > 0$, Rechenoperation. Die Multiplikation $B(VS)$ kostet dann wiederum $C_{mm}mk^2$

Zusammen erhalten wir:

$$\begin{aligned}\mathcal{N}_{trsvd}(n,m,k) &\leq && 2C_{qr}k^2n + (C_{mm}+C_{svd})k^3 + 2C_{mm}nk^2\\ &\overset{C_{mm}+C_{svd}=:C_2}{=} && 2C_{QR}k^2n + 2C_{mm}nk^2 + C_2k^3\\ &\overset{2C_{QR}+2C_{mm}+C_2=:C_{trsvd}}{\leq} && C_{trsvd}k^2n.\end{aligned}$$

■

2.2.3 Kürzung mittels Eigenwertiteration

Wie bei der ersten Kürzung haben wir hier einen Teil, den wir mittels eines externen Algorithmus berechnen. Bei Algorithmus 4 ist dies die Lösung des Eigenwertproblems.

Bemerkung 2.2.6 (Eigenvektorberechnung) *In [17] wird die Komplexität zur Berechnung der Eigenvektoren einer Matrix $M \in \mathbb{R}^{n\times n}$ mit $\mathcal{O}(n^3)$ angegeben. Wir schätzen den Aufwand mit*

$$\mathcal{N}_{ev}(n) \leq C_{ev}n^3 \tag{2.37}$$

ab.

Lemma 2.2.7 *Die Kürzung einer Rang-k-Matrix $R = AB^T \in \mathbb{R}^{n\times m}, n > m$ mittels Algorithmus 4 hat einen Aufwand von*

$$\mathcal{N}_{trev}(n,m,k) \leq C_{trev}nk^2. \tag{2.38}$$

Beweis:

- In Zeile 2 kostet die Matrix-Matrix-Multiplikation $C_{mm}nk^2$.
- In Zeile 3 kostet die Matrix-Matrix-Multiplikation $C_{mm}mk^2$.

- In Zeile 4 kostet die Matrix-Matrix-Multiplikation $C_{mm}k^3$.
- In Zeile 5 kostet die Lösung des Eigenwertproblems nach Bemerkung $C_{ev}k^3$.
- In Zeile 6 kostet die Matrix-Matrix-Multiplikation $C_{mm}mk^2$.
- Die Schleife von Zeile 7 bis 12 wird zweimal ausgeführt. Die Kosten pro Schleifendurchlauf sind $C_{mm}mk^2$ (Zeile 8), $C_{mm}k^3$ (Zeile 9), $C_{mm}mk^2$ (Zeile 10) und $C_{qr}k^2m$ (Zeile 11).
- In Zeile 13 kostet die Matrix-Matrix-Multiplikation $C_{mm}mk^2$.
- In Zeile 14 kostet die Matrix-Matrix-Multiplikation $C_{mm}nk^2$.
- Für Zeile 15 benötigen wir die Normen der Spalten. Dies kostet $C_{no}n$.
- In den Zeilen 16 und 17 werden nur noch Daten umkopiert, nach unserer Annahme entstehen dadurch keine Kosten.

Wir erhalten mit $n \geq m$

$$\begin{aligned}
\mathcal{N}_{trev}(n,m,k) &\overset{}{\leq} && 9C_{mm}nk^2 + 2C_{qr}k^2m + C_{ev}k^3 + C_{no}n \\
&\overset{9C_{mm}+2C_{qr}=:C_{tr2}}{=} && C_{tr2}nk^2 + C_{ev}k^3 + C_{no}n \\
&\overset{k\ll n, C_{trev}:=C_{tr2}+C_{ev}+C_{no}}{\leq} && C_{trev}nk^2.
\end{aligned}$$

■

Bemerkung 2.2.8 *Vergleichen wir die Konstanten $C_{trsvd} = 2C_{qr} + 3C_{mm} + C_{svd}$ und $C_{trev} = 9C_{mm} + 2C_{qr} + C_{ev} + C_{no}$, so können wir erwarten, dass der Kürzungsalgorithmus mittels Eigenvektoren langsamer sein wird als der Kürzungsalgorithmus mittels Singulärwertzerlegung. Hierbei ist zu beachten, dass in der Regel die Konstante der Singulärwertzerlegung C_{svd} größer sein wird als die Konstante C_{ev} für das Eigenwertproblem. Diese Konstanten sind aber nur für die kleinen Teilprobleme zuständig, wodurch wir sie bei dem Vergleich vernachlässigen können.*

2.2.4 Numerische Tests

Die beiden Algorithmen wurden auf einer Sun Workstation mit 2 Dual-Opterons 2218 und 10GB Arbeitsspeicher getestet. Wir fassen die Laufzeiten in einer Tabelle zusammen, wobei ϵ so gewählt wurde, dass eine Rang-k-Matrix auf den Rang $k/2$ gekürzt wurde. Der Rang wurde dabei so gewählt, dass $k/2$ eine ganze Zahl ist.
In der Tabelle steht *trsvd* für den Kürzungsalgorithmus mittels Singulärwertzerlegung und *trev* für den zweiten Kürzungsalgorithmus. Wir geben in diesen Spalten die Laufzeit in Sekunden an.

Zeilen	Spalten	Rang	*trsvd*	*trev*
10000	10000	10	0.01824s	0.03272s
10000	10000	20	0.07619s	0.17219s
10000	10000	50	0.52855s	1.14667s
10000	10000	100	2.35754s	4.70319s
50000	50000	10	0.13232s	0.26267s
50000	50000	20	0.53078s	1.06204s
50000	50000	50	3.08269s	6.43463s
50000	50000	100	12.50586s	26.20280s
100000	100000	10	0.36341s	0.70241s
100000	100000	20	1.18842s	2.69789s
100000	100000	50	7.37138s	16.35191s
100000	100000	100	26.36692s	61.95885s

Die numerischen Ergebnisse zeigen, das unsere Kürzung mittels Singulärwertzerlegung nur die Hälfte der Laufzeit benötigt, wie die Kürzung mittels der Lösung des Eigenwertproblems. Wir haben schon bei der Komplexitätsanalyse gesehen, dass die Konstante bei Kürzung mittels Singulärwertzerlegung kleiner ist (siehe Bemerkung 2.2.8).
Wir können auch sehen, dass zu einer Verdopplung der Anzahl der Zeilen ungefähr eine Verdopplung der Laufzeit von beiden Algorithmen führt. Ebenso sehen wir, dass sich die Laufzeit vervierfacht, wenn wir den Rang der Rang-k-Matrix verdoppeln, was sich aus dem Term k^2 der Komplexitätsabschätzung ergibt.

Kapitel 3

Parallele Kürzung

In diesem Kapitel wollen wir die parallelen Algorithmen definieren und bezüglich ihres Laufzeitverhaltens untersuchen.

3.1 Parallele Performance

Zur Bewertung und späteren Analyse der parallelen Algorithmen benötigen wir Performancekennzahlen. Die erste benötigte Kennzahl ist die Laufzeit eines Algorithmus.

Laufzeit 3.1.1 (Laufzeit [2]) *Gegeben seien ein Problem, das von endlich vielen Parametern $n_1, n_2, \ldots, n_q$ abhängt, und ein paralleler Algorithmus zur Lösung dieses Problems. Zusätzlich haben wir einen Parallelrechner mit p Prozessoren.*

- *$T(n_1, n_2, \ldots n_q)$ ist die Laufzeit des besten sequentiellen Algorithmus für das Problem auf einem Prozessor des Parallelrechners.*
- *$T_p(n_1, n_2, \ldots, n_q)$ ist die Zeit vom Start des parallelen Algorithmus bis zur Beendigung auf allen beteiligten Knoten. Diese Zeit nennen wir parallele Laufzeit.*

Mittels der Laufzeit können wir nun die Leistungsmaße definieren.

Definition 3.1.2 (Speedup [30]) *Der Speedup $S_p(n_1, n_2, \ldots, n_q)$ bei der Benutzung von p Prozessoren ist durch das Verhältnis der sequentiellen und parallelen Laufzeit definiert:*

$$S_p(n_1, n_2, \ldots, n_q) = \frac{T(n_1, n_2, \ldots, n_q)}{T_p(n_1, n_2, \ldots, n_q)}. \tag{3.1}$$

Wir streben einen Speedup von $S_p(n_1, n_2, \ldots, n_q) \approx p$ an. Dieser Speedup wird auch linearer Speedup oder perfekter Speedup genannt ([30]). In der Praxis ist es möglich, einen superlinearen Speedup zu erreichen, der durch die höhere Speicherbandbreite und den größeren Cachespeicher zustandekommen kann (siehe auch [30, Kapitel 1] oder [29, Kapitel 1]).

Durch Kommunikations- und Verwaltungskosten, die bei parallelen Algorithmen auftreten, erreichen wir in der Regel keinen linearen Speedup.
Mit dem Speedup können wir nun auch die parallele Effizienz (kurz: Effizienz) definieren.

Definition 3.1.3 (Parallele Effizienz [30]) *Die parallele Effizienz E_p ist durch*

$$E_p(n_1, n_2, \ldots, n_q) := \frac{S_p(n_1, n_2, \ldots, n_q)}{p} = \frac{T(n_1, n_2, \ldots, n_q)}{pT_p(n_1, n_2, \ldots, n_q)} \tag{3.2}$$

definiert.

Falls ein Algorithmus einen linearen Speedup hat, so ergibt sich für diesen Algorithmus eine Effizienz von 1. Bei einem superlinearen Speedup erhalten wir eine Effizienz, die größer als 1 ist.
Zu unserer Analyse der parallelen Algorithmen gehören die Untersuchung des Speedups anhand von Tests auf verschiedenen Rechensystemen, sowie die theoretische Untersuchung der Algorithmen. Für die theoretische Untersuchung der Algorithmen existieren verschiedene Modelle, wie zum Beispiel PRAM („parallel random access machine“, z.B. [2]), BSP („bulk synchronous parallel“, z.B. [4]) und LogP (z.B. [9, 3]). Wir werden in dieser Arbeit einen Parallelrechner anhand der folgenden Parameter modellieren.

- p: Anzahl der Berechnungseinheiten bzw. Prozessoren.
- g: Dauer, um ein Datum im Kommunikationsnetzwerk zu übertragen.
- L: Latenzzeit, die alle Verzögerungen und einmalige Kosten der Übertragung umfasst.
- s: Fließkommaoperationen pro Sekunde

Der Parameter s entfällt in unserer Laufzeitanalyse, weil die Parameter g und L nach s parametrisiert sind.
Ähnlich wie im BSP-Modell können wir unseren Algorithmus in Kommunikationsphasen und Berechnungsphasen unterteilen. Die Laufzeit jeder Phase ist das Maximum der Laufzeiten, die die einzelnen Prozessoren in dieser Phase benötigen. Falls wir in einer Berechnungsphase sind und ein Algorithmus p Prozessoren verwendet, wobei w_i, $i \in \{1, \ldots, p\}$, die Laufzeit des Prozessors i ist, dann ist die Laufzeit der Berechnungsphase durch

$$T_B := \max_{i=1}^{p} w_i \tag{3.3}$$

definiert. Seien c_i, $i \in \{1, \ldots, p\}$, die Übertragungszeiten der Daten für die Prozessoren. Damit erhalten wir die Laufzeit der Kommunikationsphase durch:

$$T_C := \max_{i=1}^{p} c_i + L. \tag{3.4}$$

In L werden die Kosten zusammengefasst, die nur einmal pro Kommunikationsphase auftreten.

Die Gesamtlaufzeit erhalten wir durch Summation der Laufzeiten aller Kommunikations- und Berechnungsphasen des Algorithmus. Das heisst, besteht ein Algorithmus aus den Kommunikationsphasen

$$T_C^1, \ldots, T_C^n \tag{3.5}$$

und den Berechnungsphasen

$$T_B^1, \ldots, T_B^m, \tag{3.6}$$

dann ist die Gesamtlaufzeit durch

$$T := \sum_{i=1}^{n} T_C^i + \sum_{i=1}^{m} T_B^i \tag{3.7}$$

gegeben.

Bemerkung 3.1.4 *Wir kombinieren bei unserem Modell Eigenschaften von BSP und LogP. Die Verwendung der Kommunikationsphasen und der Berechnungsphasen stammt vom BSP-Modell. Die Modellierung der Übertragungskosten mittels Latenzzeit L und Übertragungsgeschwindigkeit g stammt vom LogP-Modell.*
Wir verwenden diese Kombination, weil wir die Algorithmen, welche wir später einführen, leicht in Berechnungs- und Kommunikationsphasen unterteilen können. Die Experimente zur Laufzeitanalyse werden wir aber auf einem Clusterrechner mit zwei unterschiedlich schnellen Netzwerken durchführen. Deshalb wollen wir die Möglichkeit haben, die Auswirkungen unterschiedlich schneller Netzwerke zu analysieren.

3.2 Programmiermodell

Die Algorithmen wurden mittels des „Message Passing Interface“-Standards (MPI) implementiert ([28]), weswegen die Algorithmen im sogenannten „multiple instruction muliple data“-Programmiermodell (kurz: MIMD [29]) angegeben sind.
Wir gehen nun davon aus, dass wir immer p Prozessoren vorliegen haben. Diese Prozessoren werden von 1 bis p durchnummeriert. Somit wird jedem Prozessor eine eindeutige Nummer zugewiesen, welche allgemein auch als Rang bezeichnet wird. Aufgrund der Datenverteilung kann es vorkommen, dass die Prozessoren unterschiedliche Aufgaben haben. Deshalb ist es nötig, dass der Algorithmus den Rang des Prozessors abfragen kann, auf dem er gerade läuft. Diese Funktion bezeichnen wir mit

myrank().

Die Funktion gibt den Rang des Prozessors zurück, auf dem die Funktion ausgeführt wird.
Zusätzlich benötigen wir noch zwei Methoden, um Daten von einem Prozessor zu einem anderen Prozessor zu übertragen. Dies geschieht mit den Prozeduren

send(send_rank, recv_rank, data)

und

receive(send_rank, recv_rank, data).

Dabei gibt send_rank den Prozessor an, der die Daten verschickt und recv_rank den Prozessor, der die Daten empfängt. Die Daten können von beliebigen Typ sein.
Bei unserer Implementierung sind die Daten immer Matrizen, Teile von Matrizen, Vektoren oder Skalare. Dabei ist darauf zu achten, dass die Prozedur send(...) die Daten verschickt und die Prozedur receive(...) diese Daten in die bei „data“angegebene Matrix oder ähnliche Struktur kopiert. Dabei werden alle Daten überschrieben. Für jeden Aufruf der Prozedur send(...) benötigen wir somit eine Prozedur receive(...) mit den gleichen Parametern für den sendenden Prozessor und den empfangenden Prozessor. Die Datenstruktur muss ebenfalls von der gleichen Größe sein.
Bei der Implementation von parallelen Algorithmen müssen wir in der Regel darauf achten, dass sich die Kommunikationen nicht gegenseitig blockieren. Wir gehen bei unseren Algorithmen davon aus, dass eine Blockierung nicht möglich ist.

3.3 Verteilungsschema

Der erste Schritt der Parallelisierung ist, die Matrizen A und B einer Rang-k-Matrix auf die Prozessoren zu verteilen.
Der Algorithmus für die QR-Zerlegung der Matrizen A und B verwendet Skalarprodukte der Spalten. Daher bietet es sich an, diese Spalten zu unterteilen und die Skalarprodukte parallel auszuführen. Zusätzlich unterteilen wir die Matrizen so, dass bei jedem Prozessor ein zusammenhängender Block der ganzen Matrix liegt. Die Rang-k-Matrizen $R \in Rk(n,m)$ erfüllen in der Regel $n, m \gg k$. Wir fordern nun zusätzlich

$$kp \leq n \quad \text{und} \quad kp \leq m. \tag{3.8}$$

Wir werden später sehen, dass es durch diese Bedingung möglich ist, die QR-Zerlegung so zu berechnen, dass die Matrix R nur auf einem Prozessor liegt.

Definition 3.3.1 *Seien ein Parallelrechner mit p Prozessoren gegeben, sowie $M \in \mathbb{R}^{n\times k}$ mit $n \geq k$ und $kp \leq n$. Wir bezeichnen mit n_i die Anzahl der Zeilen, die bei Prozessor i liegen, und n_i wird folgendermaßen definiert:*

$$n_i := \left\lfloor \frac{n}{p} \right\rfloor + c_i \quad \forall i \in \{1,\ldots,p\} \tag{3.9}$$

mit

$$c_i := \begin{cases} 1 & : \text{ falls } i \leq n - \lfloor \frac{n}{p} \rfloor p \\ 0 & : \text{ sonst} \end{cases} \tag{3.10}$$

Somit verwaltet jeder Prozessor $i \in \{1,\ldots,p\}$ einen Teil der Matrix M, den wir mit

$$M_i \in \mathbb{R}^{n_i \times k} \tag{3.11}$$

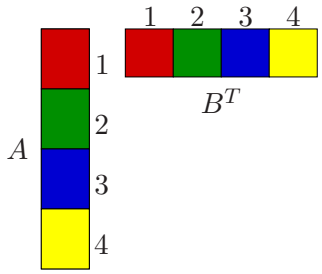


Abbildung 3.1: Verteilung einer Rang-k-Matrix auf 4 Prozessoren. Die Farben geben an, auf welchem Prozessor die Teilblöcke liegen. Prozessor 1: rot, P.2: grün, P. 3: blau, P. 4: gelb.

bezeichnen. Die Einträge sind durch

$$M_i[\tau,\sigma] := M[\tau + \sum_{j=1}^{i-1} n_j, \sigma], \quad \tau \in \{1,\dots,n_i\}, \sigma \in \{1,\dots k\} \tag{3.12}$$

definiert.

Bemerkung 3.3.2 *Für eine nach Definition 3.3.1 verteilt gespeicherte Matrix $M \in \mathbb{R}^{n\times k}$ auf p Prozessoren, gilt*

$$|n_i - n_j| \leq 1, \quad \forall i,j \in \{1,\dots,p\}. \tag{3.13}$$

Dies führt in unseren parallelen Algorithmen später zu einer guten Lastbalancierung.

Wir wollen nun die Rang-k-Matrix $AB^T \in Rk(n,m)$ verteilen. Also sind deren Faktoren A und B nach der Definition 3.3.1 verteilt, und der Prozessor $i \in \{1,\dots,p\}$ verwaltet die Matrizen $A_i \in \mathbb{R}^{n_i\times k}$ und $B_i \in \mathbb{R}^{m_i\times k}$, wobei die Einträge der Matrizen durch

$$A_i[\tau,\sigma] := A[\tau + \sum_{j=1}^{i-1} n_j, \sigma], \quad \tau \in \{1,\dots,n_i\}, \sigma \in \{1,\dots k\} \tag{3.14}$$

und

$$B_i[\tau,\sigma] := B[\tau + \sum_{j=1}^{i-1} m_j, \sigma], \quad \tau \in \{1,\dots,m_i\}, \sigma \in \{1,\dots k\} \tag{3.15}$$

gegeben sind.

Bemerkung 3.3.3 *Wir werden bei der Definition der parallelen Algorithmen sehen, dass die Bedingung (3.8) dazu führt, dass die Matrix R der parallelen QR-Zerlegung nur auf einem Prozessor berechnet und gespeichert werden muss. Dies ist aufgrund der Annahme $k \ll n,m$ sinnvoll.*

Es ist aber auch möglich, dass man R parallel berechnet und die Bedingung (3.8) verwirft. Falls R verteilt gespeichert ist und wir R später komplett auf einem Prozessor für die Singulärwertzerlegung benötigen, ist es notwendig, die verteilten Teile von R an einen Prozessor zu senden. Dadurch wird der Kommunikationsaufwand erhöht.

Es ist nicht sinnvoll, wenn jeder Teilblock in den Algorithmen eine eigene Bezeichnung hat. In unseren Algorithmen wird daher jeder Teilblock mit A_{loc} und B_{loc} bezeichnet. Die Anzahl der Zeilen werden mit n_{loc} und m_{loc} angegeben. Dies bedeutet, falls Prozessor i die Matrix A_{loc} im Algorithmus aufruft, dann ruft er genau die Teilmatrix A_i auf. Wir nehmen auch an, dass die Daten von n_i und m_i durch n_{loc} und m_{loc} bekannt sind, und werden die Berechnung nicht mehr in den Algorithmen erwähnen.

3.4 Algorithmen

In diesem Kapitel werden wir die parallelen Algorithmen definieren.

Bemerkung 3.4.1 *Die folgenden Algorithmen sind alle mit MPI implementiert. MPI enthält schon einige vorgefertigte Routinen. Neben den normalen Routinen für das Senden und Empfangen von Daten werden bei der Implementation noch folgende Routinen verwendet:*

- *bcast*
- *reduce*
- *allreduce*

Näheres über die Funktionen kann man bei [28] erfahren. Wie genau diese Routinen implementiert sind, hängt von der MPI-Version ab. Es ist zum Beispiel möglich, dass man die Broadcast-Routine, die eine Nachricht an n Knoten schickt, mit einem Aufwand von $\mathcal{O}(\log n)$ implementiert ([25]). Wir wollen aber bei der Komplexitätsanalyse eine obere Schranke für unsere Algorithmen angeben. Daher geben wie die Kommunikationen in der Regel so an, dass sie einen linearen Aufwand bezüglich der Anzahl der Prozessoren haben und verwenden nur die Funktionen send(...) und receive(...).

3.4.1 Parallele QR-Zerlegung

Die QR-Zerlegung haben wir aufgrund der Übersichtlichkeit in zwei Teile unterteilt. R wird nur für Prozessor 1 benötigt. In der Implementierung wird dies auch umgesetzt. Im Pseudocode definieren wir R aber für alle Prozessoren, weil der Code dadurch übersichtlich bleibt. Die Matrix R wird nur auf Prozessor 1 gefüllt bzw. berechnet.

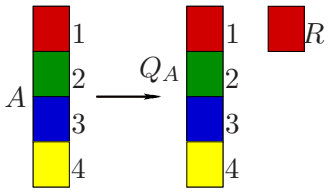


Abbildung 3.2: Verteilung der parallelen QR-Zerlegung. A ist verteilt und wird mit der Matrix Q_A überschrieben. Die Matrix R wird auf Prozessor 1 gespeichert.

Algorithmus 5 parallel_qr_decomposition(A_{loc}, R)

Benötigt: Die Matrix A_{loc} und die Matrix R.

Beschreibung: Die QR-Zerlegung wird berechnet. Q wird in A_{loc} gespeichert und R in R von Prozessor 1.

1: $R \in \mathbb{R}^{k\times k}$, $v_{loc} \in \mathbb{R}^{n_{loc}}$;
2: parallel_qr_decomposition_part1(A_{loc}, R, v_{loc});
3: parallel_qr_decomposition_part2(A_{loc}, R, v_{loc});

Der erste Teil des QR-Zerlegungsalgorithmus benötigt die Berechnung des Householdervektors, die nun parallel durchgeführt werden soll. Der Algorithmus 6 berechnet den Householdervektor parallel und gibt den Faktor β an alle Prozessoren zurück.
Die parallele Version der Householdervektor-Berechnung benötigt wie die sequentielle Version eine Teilspalte der Matrix, für welche die QR-Zerlegung berechnet werden soll. Dieser Matrix entspricht beim Kürzungsalgorithmus mittels Singulärwertzerlegung die Matrix A, respektive B. Diese Matrizen haben wir auf die p Prozessoren verteilt, womit die Spalten ebenfalls auf verschiedene Prozessoren verteilt sind.
Dem Algorithmus 1 wurden zur Bestimmung des Householdervektors immer die notwendigen Teilspalten der Matrix übergeben. Bei dem parallelen Algorithmus übergibt hingegen jeder Prozessor den gesamten Teil der entsprechenden Spalte inklusive der Größe des Teilstücks. Wir werden diese Teilstücke mit x_{loc} bezeichnen und ihre Größe wird mit n_{loc} angegeben. Dabei kann die Größe von n_{loc} auf unterschiedlichen Prozessoren unterschiedlich sein.
Wir berechnen die QR-Zerlegung nur für Matrizen $M \in \mathbb{R}^{n\times k}$, die nach der Verteilung nach Definition 3.3.1 auch Bedingung (3.8) erfüllt. Daher gilt $x_{loc} \in \mathbb{R}^{n_1}$ auf Prozessor 1, mit $n_1 \geq k$. Wie wir auch wissen, berechnen wir bei der QR-Zerlegung k Householdervektoren. Dabei hat der i-te Householdervektor die Länge $n - i + 1$. Zusammen mit $n_1 \geq k$ wissen wir, dass der Parameter, der in Zeile 6 von Algorithmus 1 x_0 zugewiesen wird, für jedes $i \in \{1, \ldots, k\}$ auf Prozessor 1 liegt. Diesen Parameter benötigen wir zur Berechnung von β. Ein weiterer benötigter Parameter ist σ aus Zeile 2 von Algorithmus 1. Dieser Parameter wird mittels eines Skalarproduktes der Teilspalte aus der Matrix bestimmt. Da die Matrix immer wieder überschrieben wird, wird für den i-ten Householdervektor das Skalarprodukt $M(i+1:n,i)^T M(i+1:n,i)$ berechnet.

Wir übergeben unserem Algorithmus die Spalte $M(1:n,i)$, die in unserem Algorithmus durch die Teilvektoren x_{loc} repräsentiert wird. Diese liegt zusätzlich verteilt auf den Prozessoren, wodurch wir das Skalarprodukt parallel berechnen müssen. Da wir nun die komplette Spalte übergeben, benötigen wir auch den Parameter i, damit wir die richtigen Teile der Vektoren x_{loc} für das Skalarprodukt verwenden. Der Householdervektor wird in den Vektor v_{loc} gespeichert, die auf jedem Prozessor vorliegen und von der aufrufenden Prozedur so initialisiert werden, dass v_{loc} und x_{loc} für jeden Prozessor immer die gleiche Größe haben. Die Berechnung von σ besteht nun aus einem parallelen Skalarprodukt, dessen Ergebnis an Prozessor 1 gesendet wird. Danach können wir β und v_{scale} auf Prozessor 1 berechnen. v_{scale} wurde im sequentiellen Algorithmus der Householderberechnung noch in $v(1)$ gespeichert (Zeile 9 oder 11 in Algorithmus 1). Die Definition des Householdervektors geschieht danach wiederum parallel.

Algorithmus 6 parallel_householdervector(v_{loc}, x_{loc}, i, n_{loc})

Benötigt: $v \in \mathbb{R}^{n_{loc}}$ ist der lokale Anteil des Householdervektors, $x \in \mathbb{R}^{n_{loc}}$ ist der lokale Teil der i-ten Spalte der Matrix A.

Beschreibung: Berechnet β und die Householdervektor v.

1: $w \in \mathbb{R}^{n_{loc}}$;
2: **if** myrank() = 1 **then**
3: $\sigma_2 := x_{loc}(i+1 : n_{loc})^T x_{loc}(i+1 : n_{loc})$;
4: **else**
5: $\sigma_2 := x_{loc}(1 : n_{loc})^T x_{loc}(1 : n_{loc})$;
6: **end if**
7: $\sigma := \sigma_2$;
8: **if** myrank() $\neq$ 1 **then**
9: send(j, 1, σ_2);
10: **else**
11: **for** $j = 2, \ldots, p$ **do**
12: receive(j, 1, σ_{recv});
13: $\sigma := \sigma + \sigma_{recv}$;
14: **end for**
15: **end if**
16: **if** myrank() = 1 **then**
17: $\beta := 0$;
18: **else**
19: $x_0 := x_{loc}(i)$;
20: $\mu := \sqrt{x_0 \cdot x_0 + \sigma}$;
21: **if** $x_0 \leq 0$ **then**
22: $v_{loc}(i) := x_0 - \mu$;
23: **else**
24: $v_{loc}(i) := -\frac{\sigma}{x_0 - \mu}$;
25: **end if**
26: $\beta := \frac{2 v_{loc}(i)^2}{\sigma + v_{loc}(i)^2}$;
27: $v_{scale} := v_{loc}(i)$;
28: **end if**
29: **if** myrank() = 1 **then**
30: **for** $j = 2, \ldots, p$ **do**
31: send(1, j, v_{scale});
32: send(1, j, β);
33: **end for**
34: **else**
35: receive(1, j, v_{scale});
36: receive(1, j, β);
37: **end if**
38: **if** myrank() = 1 **then**
39: $v(i : n_{loc}) := \frac{1}{v_{scale}} x_{loc}$;
40: $v(i) := 1$;
41: **else**
42: $v := \frac{1}{v_{scale}} x_{loc}$;
43: **end if**
44: **return** β;

Bemerkung 3.4.2 (Erklärung des Algorithmus 6)

- *In den Zeilen 2 bis 6 werden die lokalen Skalarprodukte für σ berechnet. Falls in Zeile 9 der Fall $i+1 > n_{loc}$ auftritt, dann wird das Skalarprodukt auf* 0 *gesetzt. Dieser Fall tritt auf, wenn für die Matrix $M \in \mathbb{R}^{n\times k}$, für die die QR-Zerlegung berechnet werden soll, $kp = n$ gilt.*
- *In den Zeilen 7 bis 15 werden die Ergebnisse der lokalen Skalarprodukte von allen Prozessoren an Prozessor* 1 *geschickt, welcher die lokalen Ergebnisse zusammenaddiert. Danach liegt σ auf Prozessor* 1 *vor.*
- *In den Zeilen 16 bis 28 werden die Parameter v_{scale} und β nur auf Prozessor* 1 *berechnet.*
- *Diese Parameter v_{scale} und β werden nun in den Zeilen 29 bis 37 von Prozessor* 1 *an die anderen Prozessoren verteilt.*
- *In den Zeilen 38 bis 43 wird der Householdervektor parallel berechnet.*
- *Die Zeile 44 gibt β an allen Prozessoren zurück.*

Der erste Teil des parallelen Algorithmus für die QR-Zerlegung einer Matrix $A \in \mathbb{R}^{n\times k}$ behandelt die Berechnung der Householdervektoren v_i, $i \in \{1, \ldots, k\}$, und die Berechnung von R. Dabei werden die Householdervektoren wie beim sequentiellen Algorithmus in der Matrix A zwischengespeichert. Wir werden ausnutzen, dass die Householdervektoren passend zur Zerlegung der Matrix A verteilt sind. Die Matrix R wird dabei nur auf Prozessor 1 gespeichert.
Der zweite Teil des Algorithmus bestimmt dann aus diesen zwischengespeicherten Householdervektoren die Matrix Q, welche wieder in der Matrix A gespeichert wird, wobei wir beachten müssen, dass die Matrix A nun auf verschiedene Prozessoren verteilt ist.

Algorithmus 7 parallel_qr_decomposition_part1(A_{loc}, R, v_{loc})

Benötigt: Die Matrix $A_{loc} \in \mathbb{R}^{n_{loc} \times k}$, den Vektor $v_{loc} \in \mathbb{R}^{n_{loc}}$ und die Matrix $R \in \mathbb{R}^{k \times k}$.

1: $s, s_2, s_{save} \in \mathbb{R}^k, \beta \in \mathbb{R}, v_{loc} \in \mathbb{R}^{n_{loc}}$;
2: **for** $i = 1, \ldots, k$ **do**
3: $\beta :=$ parallel_householdervector(v_{loc}, $A_{loc}(1 : n_{loc}, i)$, i, n_{loc});
4: **for** $j = i, \ldots, k$ **do**
5: **if** myrank() $= 1$ **then**
6: $s(j) := \beta v_{loc}(i : n_{loc})^T A_{loc}(i : n_{loc}, j)$;
7: **else**
8: $s(j) := \beta v_{loc}(1 : n_{loc})^T A_{loc}(1 : n_{loc}, j)$;
9: **end if**
10: **end for**
11: $s_{save} := s$
12: **for** $j = 1, \ldots, p$ **do**
13: send(myrank(), j, s_{save});
14: receive(j, myrank(), s_2);
15: $s := s + s_2$;
16: **end for**
17: **for all** $j = i, \ldots, k$ **do**
18: **if** myrank() $= 1$ **then**
19: $A_{loc}(i : n_{loc}, j) := A_{loc}(i : n_{loc}, j) + s(j) v_{loc}(i : n_{loc})(i : n_{loc})$;
20: **else**
21: $A_{loc}(1 : n_{loc}, j) := A_{loc}(1 : n_{loc}, j) + s(j) v_{loc}(1 : n_{loc})(1 : n_{loc})$;
22: **end if**
23: **end for**
24: **if** myrank() $\neq 1$ **then**
25: $A_{loc}(1 : n_{loc}, i) := v_{loc}(1 : n)$;
26: **end if**
27: **if** myrank() $= 1$ **then**
28: $A_{loc}(i + 1 : n_{loc}, i) := v_{loc}(i + 1 : n)$;
29: $R(i, i : k) := A(i, i : k)$;
30: $A_{loc}(i, i : k) := 0$;
31: $A_{loc}(i, i) := \beta$;
32: **end if**
33: **end for**

Bemerkung 3.4.3 (Erklärung des Algorithmus 7) *Durch Algorithmus 5 wird die verteilte Matrix A bereitgestellt, für die die QR-Zerlegung berechnet werden soll, sowie der Speicherplatz für die Matrix R. Außerdem wird mittels v_{loc} der Speicherplatz bereitgestellt, der für die Berechnung des Householdervektors benötigt wird. Die Verteilung der v_{loc} auf den Prozessoren stimmt mit der Verteilung der Spalten der Matrix A überein. Der Algorithmus besteht aus einer Schleife, die k-mal ausgeführt wird (Zeilen 4 bis 33). In dieser Schleife passiert folgendes:*

- *Die Zeile 3 berechnet den Householdervektor mittels Algorithmus 6. Der*

Householdervektor liegt danach verteilt in v_{loc} vor. Der Parameter β ist auf allen Prozessoren vorhanden.

- *Die Zeilen 4 bis 23 wenden die Householderreflexion des i-ten Schleifendurchlaufs, welche durch den Householdervektor und β bestimmt sind, auf die Matrix A an. Der erste Schritt dabei ist, die Skalare s_j, $j \in \{i, \ldots, k\}$, (siehe (2.15)) zu bestimmen, welche sich aus einem Skalarprodukt des Householdervektors mit Teilspalten der Matrix A ergeben. Diese Spalten sind verteilt, weshalb wir zuerst die lokalen Skalarprodukte berechnen (Zeilen 4 bis 10). Diese lokalen Ergebnisse werden danach von allen Prozessoren an alle anderen Prozessoren gesendet und auf allen Prozessoren aufsummiert (Zeilen 11 bis 16). Danach liegen die Skalare s_j auf allen Prozessoren vor. Der nächste Schritt ist, die Vektoraddition (2.16) zu berechnen. Dies geschieht in den Zeilen 17 bis 23. Wir können diese Vektoraddition parallel ausführen, denn die Skalare sind den Prozessoren bekannt und die Spalten der Matrix und der Householdervektor sind passend auf die Prozessoren verteilt.*

- *Die Zeilen 24 bis 32 dienen dazu, die i-te Zeile der oberen Dreiecksmatrix R zu speichern (Zeile 29), sowie β und den Householdervektor in A zu speichern (Zeilen 25, 28 und 31).*

Algorithmus 8 parallel_qr_decomposition_part2(A_i, R, v_{loc})

Benötigt: Die Matrix A_{loc}, den Vektor v_{loc} und die Matrix R.

1: $s \in \mathbb{R}^k, \beta \in \mathbb{R}$;
2: **for** $i = k, \ldots, 1$ **do**
3: **if** myrank() $= 1$ **then**
4: $\beta := A(i, i)$;
5: $v_{loc}(i) := 1$;
6: $v_{loc}(i+1, n_{loc}) := A_{loc}(i+1 : n_{loc}, i)$;
7: $A_{loc}(i, i) := 1$;
8: $A_{loc}(i+1 : n_{loc}, i) := 0$;
9: **end if**
10: **if** myrank() $\neq 1$ **then**
11: $v_{loc}(1, n_{loc}) := A_{loc}(1 : n_{loc}, i)$;
12: $A_{loc}(1 : n_{loc}, i) := 0$;
13: **end if**
14: **if** myrank() $= 1$ **then**
15: **for** $j = 2, \ldots, p$ **do**
16: send(1, j, β);
17: **end for**
18: **else**
19: receive(1, j, β);
20: **end if**
21: **for** $j = i, \ldots, k$ **do**
22: **if** myrank() $= 1$ **then**
23: $s(j) := \beta v_{loc}(i : n_{loc})^T A_{loc}(i : n_{loc}, j)$;
24: **else**
25: $s(j) := \beta v_{loc}(1 : n_{loc})^T A_{loc}(1 : n_{loc}, j)$;
26: **end if**
27: **end for**
28: $s_{save} := s$
29: **for** $j = 1, \ldots, p$ **do**
30: send(myrank(), j, s_{save});
31: receive(j, myrank(), s_2);
32: $s := s + s_2$;
33: **end for**
34: **for** $j = i, \ldots, k$ **do**
35: **if** myrank() $= 1$ **then**
36: $A_{loc}(i : n_{loc}, j) := A_{loc}(i : n_{loc}, j) - s(j) v_{loc}(i : n_{loc})(i : n_{loc})$;
37: **else**
38: $A_{loc}(1 : n_{loc}, j) := A_{loc}(1 : n_{loc}, j) - s(j) v_{loc}(1 : n_{loc})(1 : n_{loc})$;
39: **end if**
40: **end for**
41: **end for**

Bemerkung 3.4.4 (Erklärung des Algorithmus 8) *Algorithmus 5 liefert wieder den Speicherplatz für den Householdervektor, sowie die Matrix A. Dieser Algorithmus besteht wieder aus einer Schleife, die k-mal ausgeführt wird (Zeilen 2 bis 41), die aber in umgekehrter Reihenfolge zu Algorithmus 7 durchlaufen wird. In dieser Schleife wird die Matrix Q folgendermaßen berechnet:*

- *In den Zeilen 3 bis 20 wird der Householdervektor aus der Matrix A ausgelesen (Zeilen 6 und 11). β wird auf Prozessor* 1 *ausgelesen (Zeile 4) und durch die Zeilen 14 bis 20 an die anderen Prozessoren verteilt.*
- *Die Zeilen 15 bis 40 sind identisch (bis auf die Ausführungsreihenfolge der Schleife) mit den Zeilen 4 bis 23 aus Algorithmus 7 und berechnen nun die Matrix Q, die in A gespeichert wird.*

Das Resultat des Algorithmus 5 ist die QR-Zerlegung der Matrix A, wobei die Matrix Q in A gespeichert wird und somit verteilt auf den Prozessoren liegt, und die Matrix R, die nur auf Prozessor 1 gespeichert ist.

3.4.2 Parallele Kürzung mittels Singulärwertzerlegung

Der Algorithmus 9 berechnet die Kürzung einer Rang-k-Matrix parallel unter Verwendung der Singulärwertzerlegung. Der Algorithmus nutzt den parallelen QR-Zerlegungsalgorithmus.

Algorithmus 9 parallel_truncate_rkmatrix_svd(A_{loc}, B_{loc}, ϵ)

Benötigt: $A_{loc} \in \mathbb{R}^{n_{loc} \times k}, B_{loc} \in \mathbb{R}^{m_{loc} \times k}, \epsilon \geq 0$. Die Matrix A_{loc}, B_{loc} sind die verteilten Teilmatrizen von A und B.

Beschreibung: Der Algorithmus berechnet die Kürzung der Rang-k-Matrix, sodass der Fehler ϵ eingehalten wird. Die gekürzte Matrix wird in die verteilten Matrizen A und B geschrieben.

1: $R_A, R_B \in \mathbb{R}^{k \times k}$;
2: parallel_qr_decomposition(A_{loc}, R_A);
3: parallel_qr_decomposition(B_{loc}, R_B);
4: **if** myrank() $= 1$ **then**
5: $R := R_A R_B^T$;
6: svd(R,U,S,V);
7: $k_{new} := k$;
8: **for** $i = k, \ldots, 1$ **do**
9: **if** $S(i,i) < \epsilon$ **then**
10: $k_{new} := i$;
11: **end if**
12: **end for**
13: $R_A := U(1 : k_{new}, 1 : k)$;
14: $R_B := V(1 : k_{new}, 1 : k)S(1 : k_{new}, 1 : k_{new})$;
15: **end if**
16: **if** $myrank() = 1$ **then**
17: **for** $j = 2, \ldots, p$ **do**
18: send(1, j, R_A);
19: send(1, j, R_B);
20: **end for**
21: **else**
22: receive(1, j, R_A);
23: receive(1, j, R_B);
24: **end if**
25: $A_{loc} := A_{loc} R_A$;
26: $B_{loc} := B_{loc} R_B$;

Bemerkung 3.4.5 (Erklärung des Algorithmus 9) *Der Algorithmus 9 berechnet die Singulärwertzerlegung einer Rang-k-Matrix. Die Faktoren A und B sind, wie in Definition 3.3.1 beschrieben, verteilt. Zusätzlich soll die Kürzung eine Bestapproximation in der Spektralnorm sein, die den Fehler ϵ nicht überschreitet. Der Algorithmus arbeitet nun folgende Phasen ab:*

- *Die Zeilen 2 und 3 berechnen die QR-Zerlegungen von A und B, wobei R_A und R_B nur auf Prozessor 1 gespeichert sind.*
- *Die Zeilen 4 bis 15 werden nur von Prozessor 1 ausgeführt. Dort werden die Singulärwerte der Matrix $R := R_A R_B^T$ (Zeile 6), sowie der Rang k_{new} der gekürzten Rang-k-Matrix berechnet (Zeilen 7 bis 12). In den Zeilen 13 und 14 werden die Matrizen definiert, die wir dann mit der Matrix A und B lokal multiplizieren.*

- *Diese Matrizen werden in den Zeilen 16 bis 24 von Prozessor* 1 *an die anderen Prozessoren verteilt.*
- *In den Zeilen 25 und 26 werden die neue Faktoren der nun gekürzten Matrix berechnet.*

Als Resultat erhalten wir eine auf den Rang k_{new} gekürzte Rang-k-Matrix, welche verteilt auf den Prozessoren gespeichert ist.

3.4.3 Parallele Kürzung mittels Eigenwertiteration

Aufgrund der Übersichtlichkeit werden wir den Algorithmus wieder zerlegen. Diesmal zerlegen wir ihn in drei Teile, wobei der erste Teil die Lösung des Eigenwertproblems ermittelt und an alle Prozessoren verteilt. Im zweiten Teil findet man die Orthogonale Iteration, und im dritten Teil werden die Singulärwerte und die gekürzten Matrizen berechnet.

Algorithmus 10 parallel_truncate_rkmatrix_evp(A_{loc}, B_{loc}, ϵ)

Benötigt: $A_{loc} \in \mathbb{R}^{n_{loc} \times k}, B_{loc} \in \mathbb{R}^{m_{loc} \times k}, \epsilon \geq 0$. Die Matrix A_{loc}, B_{loc} sind die verteilten Teilmatrizen von A und B.

Beschreibung: Der Algorithmus berechnet die Kürzung der Rang-k-Matrix, sodass der Fehler ϵ eingehalten wird. Die gekürzte Matrix wird in die verteilten Matrizen A und B geschrieben.

1: $G_A \in \mathbb{R}^{k \times k}, \tilde{B}_{loc} \in \mathbb{R}^{m_{loc} \times k}$;
2: eigenvalueproblem(A_{loc}, B_{loc}, $\tilde{B}_{loc}$, G_A);
3: iteration(A_{loc}, B_{loc}, $\tilde{B}_{loc}$, G_A);
4: norms_and_truncation(A_{loc}, B_{loc}, $\tilde{B}_{loc}$,ϵ);

Bemerkung 3.4.6 (Erklärung des Algorithmus 10) *Dem Algorithmus 10 wird eine verteilte Rang-k-Matrix mit den Faktoren $A \in \mathbb{R}^{n \times k}$ und $B \in \mathbb{R}^{m \times k}$ übergeben. Diese beiden Faktoren sind nach Definition 3.3.1 verteilt. Zusätzlich werden die Matrizen $G_A \in \mathbb{R}^{k \times k}$ und $\tilde{B} \in \mathbb{R}^{m \times k}$ in den drei Teilen des Algorithmus benötigt. G_A ist auf jedem Prozessor gespeichert. $\tilde{B}$ hingegen ist wieder nach Definition 3.3.1 auf den Prozessoren verteilt und ihre Teile auf den einzelnen Prozessoren werden mit $\tilde{B}_{loc}$ bezeichnet.*
Der Algorithmus ruft nacheinander die drei Algorithmen 11, 12 und 13 auf. Als Ergebnis erhalten wir eine gekürzte Rang-k-Matrix, die die Bestapproximation in der Spektralnorm bezüglich $\epsilon > 0$ mit minimalem Rang erfüllt.

Der Algorithmus 11 berechnet die Eigenvektoren von dem Eigenwertproblem (2.22).

Algorithmus 11 eigenvalueproblem(A_{loc}, B_{loc}, $\tilde{B}_{loc}$, G_A)

1: $G, G^A_{save}, G^B_{save}, G_A, G_B, , G_2, R, X \in \mathbb{R}^{k\times k}$;
2: $G_A := A^T_{loc} A_{loc}$;
3: $G_B := B^T_{loc} B_{loc}$;
4: **if** myrank() $\neq 1$ **then**
5: send(j, 1, G_B);
6: **else**
7: **for** $j = 2, \ldots, p$ **do**
8: receive(j, 1, G^B_{save});
9: $G_B := G_B + G^B_{save}$;
10: **end for**
11: **end if**
12: $G^A_{save} := G$
13: **for** $j = 1, \ldots, p$ **do**
14: send(myrank(), j, G^A_{save});
15: receive(j, myrank(), G_2);
16: $G_A := G_A + G_2$;
17: **end for**
18: **if** myrank() = 1) **then**
19: $G := G_A G_B$;
20: evp(G, X);
21: **end if**
22: **if** myrank() = 1 **then**
23: **for** $j = 2, \ldots, p$ **do**
24: send(1, j, X);
25: **end for**
26: **else**
27: receive(1, j, X);
28: **end if**
29: $\tilde{B}_{loc} := B_{loc} X^T$;

Bemerkung 3.4.7 (Erklärung des Algorithmus 11) *Dem Algorithmus werden die verteilten Teilmatrizen der Rang-k-Matrizen übergeben, sowie die Matrix G_A, die in diesem Algorithmus gefüllt und in Algorithmus 12 wiederverwendet wird. Die Matrix $\tilde{B}$ bzw. die verteilten Teilmatrizen $\tilde{B}_{loc}$ werden mit den Eigenvektoren von $(AB^T)^T AB^T$ gefüllt.*

- *Die Zeilen 2 bis 17 berechnen die Matrizen $A^T A$ und $B^T B$, für die das $k \times k$-Eigenwertproblem gelöst wird. Die Matrizen A und B sind verteilt, deswegen berechnen wir in den Zeilen 2 und 3 die lokalen Teilprodukte von $A^T A$ bzw. $B^T B$. Die Teilprodukte von $B^T B$ werden durch die Zeilen 4 bis 11 von allen Prozessoren an Prozessor 1 gesendet, der die Teilergebnisse aufsummiert. In den Zeilen 12 bis 17 werden die Teilergebnisse von $A^T A$ von allen Prozessoren an alle anderen Prozessoren gesendet und auf allen Prozessoren aufsummiert. Somit ist $A^T A$ auf allen Prozessoren vorhanden, $B^T B$ dagegen nur auf Prozessor 1.*

- *Die Zeilen 18 bis 21 berechnen die Matrix $A^T AB^T B$ (Zeile 19) und lösen das Eigenwertproblem (Zeile 20).*
- *Die Lösung des Eigenwertproblems wird durch die Zeilen 22 bis 28 von Prozessor 1 an alle anderen Prozessoren gesendet.*
- *In Zeile 29 werden die Eigenvektoren von $(AB^T)^T AB^T$ parallel berechnet und in $\tilde{B}$ gespeichert.*

Der zweite Teil (Algorithmus 12) berechnet die Orthogonale Iteration.

Algorithmus 12 iteration(A_{loc}, B_{loc}, $\tilde{B}_{loc}$, G_A)

```
1: C, C_2, C_save, R, ∈ ℝ^{k×k};
2: for i = 1, 2 do
3:    C := B_loc^T B̃_loc;
4:    C_save := C;
5:    for j = 1, ..., p do
6:       send(myrank(), j, C_save);
7:       receive(j, myrank(), C_2);
8:       C := C + C_2;
9:    end for
10:   C_2 := G_A C;
11:   B̃_loc := B_loc C_2;
12:   parallel_qr_decomposition(B̃_loc, R);
13: end for
```

Bemerkung 3.4.8 (Erklärung des Algorithmus 12) *Der Algorithmus 12 berechnet die Orthogonale Iteration, wobei die Matrix $G_A = A^T A$ wiederverwendet wird. Die Iteration umfasst die Zeilen 2 bis 13 und besteht aus folgenden Schritten:*

- *Zeile 3 berechnet das Produkt $B^T \tilde{B}$, wobei lokale Teilergebnisse berechnet und diese dann durch die Zeilen 4 bis 9 so verteilt und aufsummiert werden, dass $B^T \tilde{B} =: C$ auf allen Prozessoren vorliegen.*
- *Die Zeilen 10 und 11 werden parallel ausgeführt.*
- *In Zeile 12 wird die QR-Zerlegung mittels des parallelen Algorithmus berechnet. Dies ist möglich, weil die Matrix $\tilde{B}$ nach Definition 3.3.1 verteilt ist.*

Nach diesem Algorithmus enthält die Matrix $\tilde{B}$ die gesuchten Eigenvektoren mit der entsprechenden Genauigkeit.

Der Algorithmus 13 führt nun noch die benötigten Matrixoperationen durch, damit die Normen der Spalten von $\tilde{A}$ die Singulärwerte ergeben. Die Normen werden parallel berechnet. Die Ergebnisse der Normberechnung liegen auf allen Prozessoren vor. Wie im sequentiellen Algorithmus wählen wir dann die entsprechenden Spalten aus, wobei dies im parallelen Algorithmus auf jedem Prozessor geschieht. Danach können wir die gekürzte Rang-k-Matrix definieren.

Algorithmus 13 norms_and_truncation(A_{loc}, B_{loc}, $\tilde{B}_{loc}$, ϵ)

1: $s \in \mathbb{R}^k, G, C_2, C_{save} \in \mathbb{R}^{k\times k}, \tilde{A}_{loc} \in \mathbb{R}^{n_{loc}\times k}$;
2: $G := B_{loc}^T \tilde{B}_{loc}$;
3: $C_{save} := G$;
4: **for** $j = 1, \ldots, p$ **do**
5: send(myrank(), j, C_{save});
6: receive(j, myrank(), C_2);
7: $G := G + C_2$;
8: **end for**
9: $\tilde{A}_{loc} := A_{loc}G$
10: **for** $i = 1, \ldots, k$ **do**
11: $s[i] := \tilde{A}_{loc}(:, i)^T \tilde{A}_{loc}(:, i)$;
12: **end for**
13: $s_{save} := s$
14: **for** $j = 1, \ldots, p$ **do**
15: send(myrank(), j, s_{save});
16: receive(j, myrank(), s_2);
17: $s := s + s_2$;
18: **end for**
19: **for** $i = 1, \ldots, k$ **do**
20: $s[i] := \sqrt{(s[i])}$;
21: **end for**
22: $I = \{i \in \{1, \ldots k\} \mid \|\tilde{A}(\cdot, i)\|_2 > \epsilon\}$;
23: $A_{loc} := \tilde{A}_{loc}(i, I)$;
24: $B_{loc} := \tilde{B}_{loc}(i, I)$;

Bemerkung 3.4.9 (Erklärung des Algorithmus 13) *Dem Algorithmus werden die Matrizen A, B und $\tilde{B}$ übergeben, welche verteilt vorliegen.*

- *Die Zeilen 2 bis 9 berechnen die Matrix $\tilde{A}$, die ebenso verteilt vorliegt. Dabei werden wieder Teilergebnisse auf einzelnen Prozessoren berechnet (Zeile 2), die dann auf allen Prozessoren aufaddiert werden (Zeilen 5 bis 8). $\tilde{A}$ wird in Zeile 9 parallel berechnet.*
- *Die Zeilen 10 bis 21 berechnen die Normen der Spalten von $\tilde{A}$, sodass die Ergebnisse der Berechnung auf allen Prozessoren vorliegt.*
- *Dadurch können wir die Spaltenauswahl für die Rang-k-Matrix auf jedem Prozessor parallel ausführen (Zeile 22).*
- *Die Definition der gekürzten Rang-k-Matrix wird parallel in den Zeilen 23 und 24 berechnet. Der Rang der neuen Matrix ist $\#I$.*

3.5 Laufzeitanalyse

Die Laufzeitanalyse führen wir nach dem Modell aus Kapitel 3.2 durch. Wir werden wie in Kapitel 2 zuerst die Laufzeit der QR-Zerlegung abschätzen und danach unsere beiden Kürzungsalgorithmen behandeln.

3.5.1 Die QR-Zerlegung

Zuerst schätzen wir die Laufzeit für die Berechnung des Householdervektors ab.

Satz 3.5.1 *Die Laufzeit $T_p^{house}(n)$ des Algorithmus 6, der den Householdervektor berechnet, kann durch*

$$T_p^{house}(n) \leq C_H \frac{n}{p} + C_h p(L+g) \tag{3.16}$$

abgeschätzt werden.

Beweis: Wir gehen die Zeilen durch, die Kosten verursachen. Wie bei den sequentiellen Algorithmen gehen wir davon aus, dass Speicheroperationen kostenlos sind.

- Die Zeilen 1 bis 6 ist eine Berechnungsphase. Hier wird ein Skalarprodukt der Teilvektoren der Länge n_i auf jedem Prozessor i berechnet, wobei wir n_i nach der Definition von 3.3.1 mit

  $$n_i \leq \frac{n}{p} + 1 \quad \forall i \in \{1, \ldots, p\} \tag{3.17}$$

  abschätzen können. Wir werden in Zukunft nur $\frac{n}{p}$ als Abschätzung für n_i verwenden. Dies können wir machen, weil in Verbindung mit dieser Abschätzung immer eine Konstante auftritt, die wir entsprechend wählen, sodass die Abschätzung gilt. Es reicht in der Regel, die Konstante um 1 zu erhöhen. Wir können diese Phase mit

  $$C_{sp} \frac{n}{p} \tag{3.18}$$

  abschätzen.

- Die Zeilen 8 bis 14 bilden eine Schleife, die $k-1$-mal durchlaufen wird. Dabei besteht jeder Schleifendurchlauf aus einer Kommunikationsphase (Zeilen 9 und 12) und einer Berechnungsphase (Zeile 13). In der Kommunikationsphase verschickt jeder Prozessor $i \neq 1$ genau eine Zahl an Prozessor 1. Für den Aufwand erhalten wir somit für jede Kommunikationsphase

  $$L + g. \tag{3.19}$$

  Die Berechnungsphase besteht aus einer Addition, was einer Laufzeit für jede Berechnungsphase von

  $$1 \tag{3.20}$$

  entspricht. Insgesamt erhalten wir für diesen Codeabschnitt

  $$\sum_{i=2}^{p} (L+g+1) = (p-1)(L+g+1) \overset{g>1}{\leq} 2p(L+g). \tag{3.21}$$

- Die Zeilen 16 bis 28 entsprechen einer Berechnungsphase, wobei nur Prozessor 1 beschäftigt ist. Der Codeteil stimmt mit dem des sequentiellen Algorithmus überein und wir schätzen die Laufzeit mit C_s ab.

- Die Zeilen 29 bis 37 entsprechen einer Kommunikationsphase, in der 2 Zahlen von Prozessor 1 an die anderen Prozessoren versendet werden. Insgesamt erhalten wir
$$L + 2g(p-1). \tag{3.22}$$
- Zeile 42 stellt eine Berechnungsphase mit einer konstanten Laufzeit C_1 dar.

Insgesamt erhalten wir:

$$\begin{aligned} T_p^H(n) &\leq C_{sp}\frac{n}{p} + 2p(L+g) + C_s + L + g(p-1)2 + C_1 \\ &\overset{C_s+C_1=:C_2}{=} C_{sp}\frac{n}{p} + C_2 + 2p(L+g) + L + g(p-1)2 \\ &\leq C_{sp}\frac{n}{p} + C_2 + 4p(L+g) \\ &\overset{C_{sp}+C_2=:C_H}{\leq} C_H\frac{n}{p} + 4p(L+g) \end{aligned}$$

Setzen wir $C_h := 4$, so ist die Behauptung bewiesen.

■

Satz 3.5.2 *Die Laufzeit $T_p^{QR}(n,k)$ der parallelen QR-Zerlegung einer Matrix $A \in \mathbb{R}^{n\times k}$ mittels Algorithmus 5 kann durch*

$$T_pQR(n,k) = C_{QRP}\, k^2\frac{n}{p} + C_{QRC}\, p(L+gk) \tag{3.23}$$

abgeschätzt werden.

Beweis: Den Algorithmus für die Berechnung der QR-Zerlegung haben wir wegen der besseren Übersichtlichkeit in zwei Algorithmen aufgeteilt. Der Algorithmus 7 besteht aus einer Schleife, die k-mal ausgeführt wird.

- Zeile 3 hat nach Satz 3.5.1 einen Aufwand von $C_H\frac{n}{p} + C_hp(L+g)$.
- Berechnungsphase von Zeile 4 bis 10: Hier haben wir eine Schleife, die maximal k-mal ausgeführt wird mit einem Skalarprodukt, welches anschließend noch skaliert wird. Dies ergibt eine Laufzeit von
$$C_{ssp}k\frac{n}{p}. \tag{3.24}$$
- Von Zeile 12 bis 16 haben wir wieder eine Schleife, bei der in jedem Schleifendurchlauf eine Kommunikationsphase und eine Berechnungsphase (Zeile 15) zu finden ist. Hier sendet aber jeder Prozessor an jeden anderen Prozessor k Zahlen. Somit erhalten wir eine Laufzeit von
$$(p-1)(L+gk) \tag{3.25}$$
für die Kommunikationsphasen und
$$(p-1)C_ak \tag{3.26}$$
für die Berechnungsphasen.

- Berechnungsphase von Zeile 17 bis 23: Wir haben wieder eine Schleife der maximalen Länge k. Die Laufzeit lässt sich somit durch

$$C_{ssp}k\frac{n}{p}. \tag{3.27}$$

  abschätzen.

Zusammen erhalten wir für die Berechnungsphasen

$$\begin{aligned} T_B^1 &:= && k(C_H\frac{n}{p} + 2C_{ssp}k\frac{n}{p} + (p-1)C_ak) \\ &\overset{(p-1)k\leq k\frac{n}{p}}{\leq} && k(C_H\frac{n}{p} + (2C_{ssp}+C_a)k\frac{n}{p}) \\ &\overset{C_H+2C_{ssp}+C_a=:C_1}{\leq} && C_1k^2\frac{n}{p}. \end{aligned} \tag{3.28}$$

Für die Kommunikationsphasen gilt die Abschätzung

$$\begin{aligned} T_C^1 &:= && (p-1)(L+gk) + C_hp(L+g) \\ &\leq && p(L+gk) + C_hp(L+gk) \\ &\overset{C_{1c}:=C_h+1}{=} && C_{1c}p(L+gk). \end{aligned} \tag{3.29}$$

Der zweite Teil des Algorithmus, der die QR-Zerlegung berechnet, besteht wieder aus einer Schleife, die k-mal ausgeführt wird. Somit erhalten wir:

- Kommunikationsphase von Zeile 14 bis 20, in der Prozessor 1 jeweils eine Zahl an die anderen Prozessoren sendet. Dies ergibt $L + (p-1)g$.

- Die Zeilen 21 bis 27 beschreiben eine Berechnungsphase, die mit der Berechnungsphase in den Zeilen 4 bis 10 aus Algorithmus 7 identisch ist. Der Aufwand ist somit $C_{ssp}k\frac{n}{p}$.

- Die Zeilen 29 bis 33 sind identisch mit Zeilen 12 bis 16 aus Algorithmus 7, und somit gilt die Abschätzung

$$(p-1)(L+gk) \tag{3.30}$$

  für die Kommunikationsphasen und

$$(p-1)C_ak \tag{3.31}$$

  für die Berechnungsphasen.

- Die Zeilen 34 bis 40 stellen eine Berechnungsphase dar. Die Laufzeitabschätzung ergibt

$$C_{ssp}k\frac{n}{p}. \tag{3.32}$$

Insgesamt erhalten wir für die Berechnungsphasen

$$\begin{aligned} T_B^2 &:= k(C_{ssp}k\frac{n}{p} + (p-1)C_a k + C_{ssp}k\frac{n}{p}) \\ &\leq k(C_{ssp}k\frac{n}{p} + C_a k\frac{n}{p} + C_{ssp}k\frac{n}{p}) \\ &\overset{2C_{ssp}+C_a=:C_2}{=} C_2 k^2 \frac{n}{p}. \end{aligned} \tag{3.33}$$

Für die Kommunikationsphasen ergibt sich

$$\begin{aligned} T_C^2 &:= k((p-1)(L+gk) + L + (p-1)g) \\ &\leq 2p(L+gk). \end{aligned} \tag{3.34}$$

Insgesamt erhalten wir

$$\begin{aligned} T_p^{QR}(n,k) &\leq T_B^1 + T_B^2 + T_C^1 + T_C^2 \\ &\leq C_1 k^2 \frac{n}{p} + C_2 k^2 \frac{n}{p} + C_{1c}p(L+gk) + 2p(L+gk) \\ &\overset{C_1+C_2=:C_{QRP}}{=} C_{QRP}k^2\frac{n}{p} + C_{1c}p(L+gk) + 2p(L+gk) \\ &\overset{C_{1c}+2=:C_{QRC}}{=} C_{QRP}k^2\frac{n}{p} + C_{QRC}p(L+gk). \end{aligned} \tag{3.35}$$

■

3.5.2 Kürzung mittels Singulärwertzerlegung

Satz 3.5.3 *Die Laufzeit $T_p^{T1}(n,m,k)$ des Algorithmus 9, der eine Rang-k-Matrix AB^T, $A \in \mathbb{R}^{n\times k}$, $B \in \mathbb{R}^{m\times k}$, $n \geq m$, mittels Singulärwertzerlegung kürzt, kann durch*

$$T_p^{T1}(n,m,k) = C_{T1}\, k^2 \frac{n}{p} + C_{t1}\, k^3 + 3C_{QRC}\, p(L+gk^2) \tag{3.36}$$

abgeschätzt werden.

Beweis:

- Die Zeilen 2 und 3 haben zusammen eine Laufzeit von

$$2C_{QR}k^2\frac{n}{p} + 2C_{QRC}p(L+gk). \tag{3.37}$$

- In der Berechnungsphase von Zeile 4 bis 15 wird nur Prozessor 1 benötigt, und wir haben hier die Laufzeit

$$C_{mm}k^3 + C_{svd}k^3, \tag{3.38}$$

wie in Algorithmus 3.

- Die Zeilen 16 bis 24 stellen eine Kommunikationsphase dar, wobei Prozessor 1 die Matrizen R_A und R_B an die anderen versendet. Wir erhalten einen Aufwand von
$$L + g(2(p-1)kk_{new}). \tag{3.39}$$
- Die Zeilen 25 und 26 benötigen
$$C_{mm}\frac{n}{p}kk_{new} \tag{3.40}$$
Rechenoperationen.

Zusammen ergibt sich für den Kommunikationsanteil

$$\begin{aligned} T_p^{CT1} &\leq 2C_{QRC}p(L+gk) + L + g(2(p-1)kk_{new}) \\ &\leq 2C_{QRC}p(L+gkk_{new}) + L + g(2(p-1)kk_{new}) \\ &\leq 3C_{QRC}p(L+gkk_{new}) \qquad (3.41) \\ &\leq 3C_{QRC}p(L+gk^2). \qquad (3.42) \end{aligned}$$

Für die Berechnungsphasen erhalten wir

$$\begin{aligned} T_p^{BT1} &\leq 2C_{QR}k^2\frac{n}{p} + C_{mm}k^3 + C_{svd}k^3 + C_{mm}\frac{n}{p}kk_{new} \\ &\overset{2C_{QR}+C_{mm}=:C_{T1}}{\leq} C_{T1}k^2\frac{n}{p} + C_{mm}k^3 + C_{svd}k^3 \\ &\overset{C_{mm}+C_{svd}=:C_{t1}}{=} C_{T1}k^2\frac{n}{p} + C_{t1}k^3. \qquad (3.43) \end{aligned}$$

Insgesamt erhalten wir

$$\begin{aligned} T_p^{T1}(n,m,k) &\leq T_p^{CT1} + T_p^{BT1} \\ &\leq C_{T1}k^2\frac{n}{p} + C_{t1}k^3 + 3C_{QRC}p(L+gk^2). \qquad (3.44) \end{aligned}$$

■

3.5.3 Kürzung mittels Eigenwertiteration

Satz 3.5.4 *Die Laufzeit $T_p^{T2}(n,k)$ des Algorithmus 10, der eine Rang-k-Matrix AB^T, $A \in \mathbb{R}^{n\times k}$, $B \in \mathbb{R}^{m\times k}$, $n \geq m$ mittels der Lösung eines Eigenwertproblems kürzt, kann durch*

$$T_p^{T2}(n,m,k) = C_{T2}\,\frac{n}{p}^2 k + C_3k^3 + C_{t2}\,pk^2 + C_{TR2C}\,p(L+gk^2) \tag{3.45}$$

abgeschätzt werden.

Beweis: Dieser Beweis unterscheidet sich nicht von den vorherigen. Wir werden hier zuerst die Laufzeit der Berechnungsphasen der drei Algorithmen 11, 12 und 13 gemeinsam abschätzen. Wir geben die Zeilen mit dem Tupel (x,y) an, wobei x die Nummer des Algorithmus ist und y die Zeile.

- (11,2): $C_{mm}\frac{n^2}{p}k$
- (11,3): $C_{mm}\frac{n^2}{p}k$
- (11,16): $p(C_s k^2)$ (Faktor p wegen der Schleife in Zeile 4)
- (11,9): $p(C_s k^2)$ (Faktor p wegen der Schleife in Zeile 4)
- (11,19): $C_{mm}k^3$
- (11,20): $C_{ew}k^3$
- (12,3): $2C_{mm}\frac{n^2}{p}k$
- (12,8): $2p(C_s k^2)$
- (12,10): $2C_{mm}k^3$
- (12,11): $2C_{mm}\frac{n}{p}k^2$
- (12,12): $2C_{QR}k^2\frac{n}{p}$
- (13,2): $C_{mm}\frac{n^2}{p}k$
- (13,7): $p(C_s k^2)$
- (13,9): $2C_{mm}\frac{n}{p}k^2$
- (13,11): $C_{sp}k^2$
- (13,17): $C_s k$
- (13,20): C_1

Zusammen erhalten wir für die Terme mit $\frac{n^2}{p}k$:

$$5C_{mm}\frac{n^2}{p}k. \tag{3.46}$$

Für Terme mit $\frac{n}{p}k^2$:

$$(2C_{mm}+2C_{QR})\frac{n}{p}k^2 \overset{2C_{mm}+2C_{QR}=:C_2}{=} C_2\frac{n}{p}k^2. \tag{3.47}$$

Für die restlichen Terme erhalten wir

$$(3C_{mm}+C_{ew})k^3+(5pC_s+C_{sp})k^2+C_s k+C_1 \tag{3.48}$$

wobei wir diesen Teil einfach durch

$$C_3k^3+C_4pk^2 \tag{3.49}$$

abschätzen wollen. Zusammen erhalten wir:

$$\begin{aligned} T_p^B &\leq 5C_{mm}\frac{n}{p}^2 k + C_2\frac{n}{p}k^2 + C_3k^3 + C_4pk^2 \\ &\overset{\frac{n}{p}>k,\ 5C_{mm}+C_2=:C_{T2}}{\leq} C_{T2}\frac{n}{p}^2 k + C_3k^3 + C_4pk^2 \end{aligned} \tag{3.50}$$

Für die Kommunikationsphasen ergibt sich Folgendes, wobei wir nur die Zeilen angeben, wo etwas gesendet wird.

- (11,5): $p(L + gk^2)$
- (11,14): $p(L + gk^2)$
- (11,24): $L + (p - 1)gk^2$
- (11,6): $p(L + gk^2)$
- (11,12): $C_{QRC}p(L + gk)$
- (11,5): $p(L + gk^2)$
- (11,15): $p(L + gk)$

Zusammen erhalten wir

$$\begin{aligned} T_p^C &\leq 4p(L + gk^2) + L + (p - 1)gk^2 + p(L + gk) + C_{QRC}p(L + gk) \\ &\leq 5p(L + gk^2) + (C_{QRC} + 1)p(L + gk) \\ &\leq (C_{QRC} + 6)(L + gk^2) \\ &\overset{C_{QRC}+6=:C_{TR2C}}{=} C_{TR2C}p(L + gk^2) \end{aligned} \tag{3.51}$$

Die Kommunikationsphase und die Berechnungsphase ergeben zusammen ($C_4 = C_{t2}$):

$$\begin{aligned} T_p^{T2}(n, m, k) &\leq T_p^B +_p^C \\ &= C_{T2}\frac{n}{p}^2 k + C_3k^3 + C_{t2}pk^2 + C_{TR2C}p(L + gk^2). \end{aligned} \tag{3.52}$$

■

Wir haben nun die Laufzeit der parallelen Algorithmen mittels unseres Modells abgeschätzt. Aber wie müssen wir diese Ergebnisse für unsere Kürzung der Rang-k-Matrix interpretieren?

3.5.4 Interpretation der Laufzeitanalyse

Wir wollen zuerst die Laufzeitanalysen der beiden parallelen Kürzungsalgorithmen für eine Rang-k-Matrix $R \in Rk(n, m)$ mit $n \geq m$ vergleichen. Der Kürzungsalgorithmus mittels Singulärwertzerlegung hat nach Satz 3.5.3 eine Laufzeit von:

$$T_p^{T1}(n, k, p) = C_{T1}k^2\frac{n}{p} + C_{t1}k^3 + 3C_{QRC}p(L + gk^2).$$

Der Kürzungsalgorithmus mittels der Eigenvektoren hat nach Satz 3.5.4 eine Laufzeit von:

$$T_p^{T2}(n,k,p) = C_{T2}\frac{n}{p}^2 k + C_3 k^3 + C_{t2} p k^2 + C_{TR2C} p(L + g k^2).$$

Für die Rang-k-Matrix gilt nach unseren Annahmen immer $n, m \gg k$ und $n \geq kp$. Der Aufwand der Berechnungsphasen beider Kürzungsalgorithmen ist von der Komplexität

$$\mathcal{O}(k^2 \frac{n}{p}). \tag{3.53}$$

Der Aufwand der Kommunikationsphasen ist ebenfalls für beide Kürzungsalgorithmen von der gleichen Komplexität:

$$\mathcal{O}(p(L + g k^2). \tag{3.54}$$

Somit haben beide parallelen Kürzungsalgorithmen insgesamt wieder die gleiche Komplexität. Aber wie im sequentiellen Fall sind die Konstanten bei der Laufzeitanalyse für den parallelen Kürzungsalgorithmus mittels Eigenvektoren höher als beim Kürzungsalgorithmus mittels Singulärwertzerlegung.
Eine weitere Frage ist das Verhalten des Speedups unserer parallelen Algorithmen. Für diese Untersuchung nehmen wir an, dass die Anzahl der Zeilen n gleich der Anzahl der Spalten m ist. Zusätzlich gilt $n \gg k$ und $n \geq kp$. Unter diesen Voraussetzungen dominiert

$$\frac{n}{p}^2 k \tag{3.55}$$

den Aufwand der Berechnungsphasen unserer beiden Kürzungsalgorithmen. Daher können wir einen linearen Speedup für entsprechend große n, k erwarten.
Zusätzlich können wir erwarten, dass der Speedup für den Kürzungsalgorithmus, der die Eigenvektoren zur Kürzung verwendet, höher ausfällt als der Speedup für den anderen Algorithmus. Dies liegt daran, dass der sequentielle Anteil geringer ist.

3.6 Numerische Experimente

Wir wollen nun unserer Ergebnisse der Analyse an numerischen Experimenten verifizieren.

3.6.1 Die Computersysteme

Wir testen die Algorithmen auf einem „shared memory“-System sowie auf einem Linuxcluster mit verteilten Speicher.

1. Der Linuxcluster besteht auf folgenden Knoten:
   (a) 32 Knoten, die mit Infiniband-Netzwerk ausgestattet sind.
      - Dual-Core AMD Opteron 254 (2.8GHz) mit 16GByte Arbeitsspeicher
      - Transferrate: max. 800MB pro Sekunde
      - Latenzzeit: 5ms

   (b) 74 Knoten, die Gigabyte-Ethernet ausgestattet sind.
      - Dual AMD Opteron 250 (2.4GHz) mit 4GByte Arbeitsspeicher
      - Transferrate: max. 125 MB pro Sekunde
      - Latenzzeit: 50 - 60ms
2. Sun-Workstation mit 2 Dual-Core AMD Opteron 2218 (2.6GHz) Prozessoren und 10GByte Arbeitsspeicher. Betriebssystem: Kubuntu 7.10.

Mittels des Linuxclusters können wir für unsere Algorithmen die Auswirkung unterschiedlicher Latenzzeiten und Übertragungsgeschwindigkeiten testen.
Wir werden zuerst den Speedup auf den verschiedenen Rechensystemen für die beiden Kürzungsalgorithmen untersuchen. Zuletzt vergleichen wir dann die Laufzeiten der beiden Algorithmen.
Für den Speedup geben wir bei jeder der folgenden Abbildungen vier Grafiken an. Dabei werden Rang-k-Matrizen mit der gleichen Zeilen- und Spaltenanzahl untersucht. Die Anzahl der Spalten und Zeilen der Matrizen wird in der Legende angegeben. Die Ränge der Matrizen sind bei den Grafiken unterschiedlich. Die obere linke Grafik visualisiert den Speedup für Rang-k-Matrizen mit Rang 10, oben rechts für Rang-k-Matrizen mit Rang 20, unten links mit Rang 50 und unten rechts mit Rang 100. Auf der horizontalen Achse sind die verwendeten Prozessoren aufgetragen und auf der vertikalen Achse der Speedup. Die schwarze Linie, die in der Legende der Grafiken als „optimal“bezeichnet wird, ist eine Referenzlinie, die den optimalen Speedup darstellt, den wir mit unseren Algorithmen erreichen wollen.
Die Tests wurden größtenteils gestartet, wenn der Linuxcluster nicht von anderen Prozessen belegt war. Andere Prozesse die die Rechensysteme benutzen können die Ergebnisse verfälschen, weil sie die Übertragungsgeschwindigkeit des Netzwerks reduzieren können. Zusätzlich kann es passieren, dass man sich einzelne Prozessoren teilen muss, was ebenfalls die Laufzeit erhöhen kann. Solche Einflüsse sehen wir zum Beispiel in der Abbildung 3.6.

3.6.2 Distributed Memory Tests

Wir wollen unsere Algorithmen zuerst auf dem Linuxcluster testen. Dabei laufen die Tests mit 1, 2, 3, 4, 6, 8, 10, 12, 14 und 16 Prozessoren.

Gigabit-Netzwerk

Die Abbildung 3.3 gibt den Speedup für unseren parallelen Kürzungsalgorithmus mittels Singulärwertzerlegung auf dem Linuxcluster an, wenn wir das Gigabit-Netzwerk benutzen.
Betrachten wir zuerst das Verhalten des Speedups, wenn wir die Anzahl der verwendeten Prozessoren fixieren. Dann sehen wir in jeder der vier Grafiken aus Abbildung 3.3, dass sich der Speedup erhöht, sobald wir die Anzahl der Zeilen bzw. der Spalten erhöhen. Der Speedup erhöht sich ebenfalls, wenn wir die Anzahl der Zeilen fixieren und nur den Rang der Rang-k-Matrix erhöhen. Dies haben wir schon in der Interpretation der Laufzeitanalyse erwartet. Je größer die Matrix ist, umso höher sollte der Speedup sein, weil dann der Term $\frac{n^2}{p}k$ dominanter wird.
Betrachten wir nun das Verhalten des Speedups, wenn wir für eine Matrix die Anzahl der Prozessoren erhöhen, mit der wir sie kürzen. Dies sind genau die Linien in den Grafiken. Wir sehen dort, dass sich der Anstieg des Speedups immer mehr verringert je mehr Prozessoren für die Kürzung verwendet werden. Bei manchen Matrizen verringert sich dabei sogar der Speedup. Dies liegt an den Kommunikationskosten des Algorithmus. Diese betragen nach Satz 3.5.3

$$3C_{QRC}p(L + gk^2).$$

Besonders bei kleinen Matrizen (siehe zum Beispiel die linke obere Grafik in Abbildung 3.3 mit Rang 10) übersteigt der Aufwand der Kommunikation nun den Aufwand der Berechnungsphasen, wodurch sich der Speedup durch die Verwendung von immer mehr Prozessoren verringert. Aber selbst für viele der Matrizen mit Rang 10 erhalten wir noch einen optimalen Speedup bei der Benutzung von 3 Prozessoren.
Die Abbildung 3.4 gibt nun den Speedup auf dem Linuxcluster an, wenn wir die Rang-k-Matrizen durch unseren zweiten Kürzungsalgorithmus kürzen. Die Ergebnisse sind sehr ähnlich zu denen vom Kürzungsalgorithmus mittels Singulärwertzerlegung.
Der Speedup steigt an, je größer die Rang-k-Matrizen werden. Dies gilt wiederum, wenn man die Anzahl der Spalten oder den Rang der Matrix erhöht. Ebenso verringert sich der Anstieg, wenn wir für eine fixierte Rang-k-Matrix die Prozessorzahl für die Kürzung erhöhen. Dies ist auch bei diesem Algorithmus eine Folge der Kommunikationskosten (Satz 3.5.4):

$$C_{TR2C}p(L + gk^2).$$

Im Allgemeinen hat der zweite Kürzungsalgorithmus einen höheren Speedup. Dies liegt daran, dass die Berechnungsphase des zweiten Kürzungsalgorithmus einen höheren Aufwand hat, als die Berechnungsphase des ersten Algorithmus. Dies haben wir auch schon bei den sequentiellen Algorithmen festgestellt.

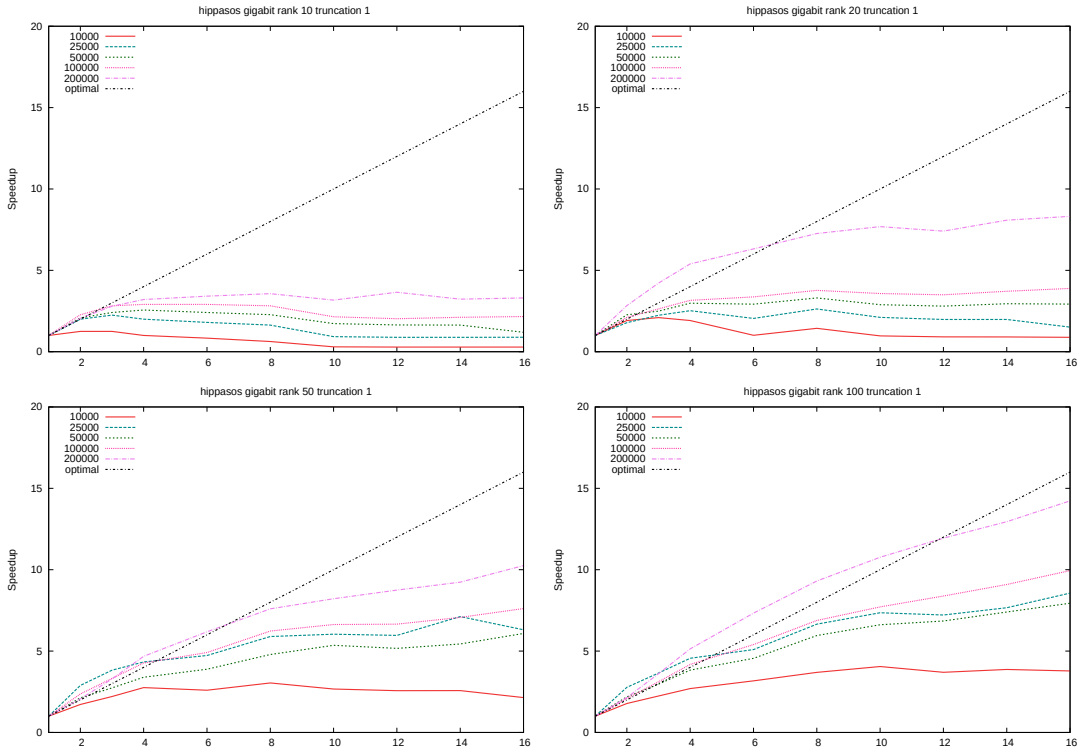


Abbildung 3.3: Test der Kürzung mittels der Singulärwertzerlegung mit Gigabit-Netzwerk

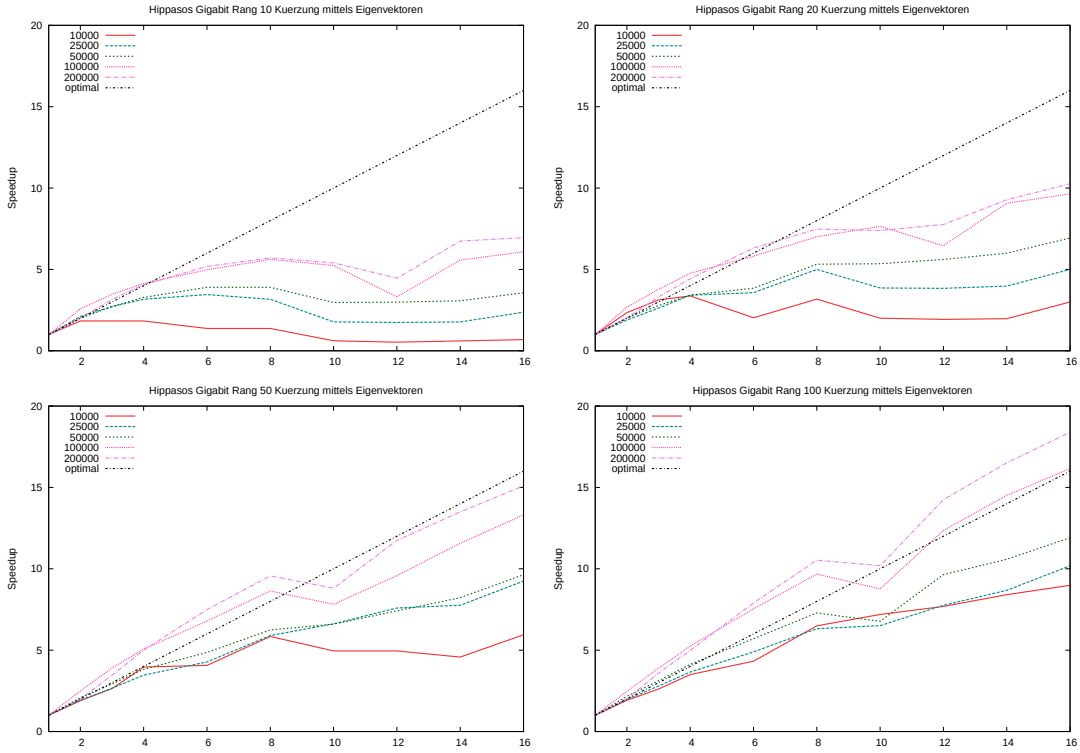


Abbildung 3.4: Test der Kürzung mittels Eigenvektoren mit Gigabit-Netzwerk

Infiniband-Netzwerk

Nun wollen wir die Kürzungsalgorithmen auf den Linuxcluster unter der Verwendung des Infiniband-Netzwerks testen. In der Abbildung 3.5 ist der Speedup dargestellt, den wir für den Kürzungsalgorithmus mittels Singulärwertzerlegung erhalten. Die Abbildung 3.6 stellt den Speedup für den Kürzungsalgorithmus mittels Eigenvektoren dar. Die Ergebnisse sind ähnlich zu den Ergebnissen der Test mittels des Gigabit-Netzwerkes. Bei beiden Algorithmen steigt der Speedup an, je größer die zu kürzenden Matrizen werden.
Der Speedup ist dank des schnelleren Infiniband-Netzwerks größer als bei der Verwendung des Gigabit-Netzwerkes. Wir erhalten bei dem Test mittels Infiniband-Netzwerk schon bei fast allen Matrizen mit Rang 50 einen optimalen Speedup oder fast optimalen Speedup für alle verwendeten Prozessoranzahlen. Bei den Tests mittels des Gigabit-Netzwerkes sehen wir diesen fast optimalen Speedup erst für Rang-k-Matrizen mit Rang 100 und dies auch nur für unseren zweiten Kürzungsalgorithmus (siehe Abbildung 3.4).
Generell können wir feststellen, dass beide Algorithmen auf Rechensystemen mit verteiltem Speicher einen optimalen Speedup erreichen können. Damit wir einen linearen Speedup erreichen, muss die Anzahl der Prozessoren in Abhängigkeit der Größe der Rang-k-Matrizen und der Geschwindigkeit des Netzwerkes gewählt werden.
Die Laufzeiten der Algorithmen für ausgewählte Rang-k-Matrizen unter der Verwendung von 16 Prozessoren, die über das Infiniband-Netzwerk kommunizieren, sind:

Zeilen	Spalten	Rang	$trsvd$	$trev$
10000	10000	50	0.07459s	0.08906s
50000	50000	50	0.26233s	0.54225s
100000	100000	50	0.55485s	1.14501s

Der Speedup des zweiten Algorithmus ist in den Tests in der Regel höher als der Speedup des Kürzungsalgorithmus mittels Singulärwertzerlegung. Wie wir aber anhand der Laufzeiten der beiden Algorithmen sehen, lohnt es sich nicht den zweiten Algorithmus zu verwenden, weil er, aufgrund seines höheren Aufwands, eine längere Laufzeit aufweist.

3.6.3 Shared Memory Tests

Bei „shared memory“-Systemen fällt für die Laufzeitanalyse der Kommunikationsteil weg, weil die Prozessoren direkt auf die Daten zugreifen können und Daten nicht über ein Netzwerk verschickt werden müssen. Daher erwarten wir für solche Systeme, dass wir einen optimalen Speedup erreichen. Die Tests auf der Sun-Workstation liefern für den Kürzungsalgorithmus mittels Singulärwertzerlegung als Ergebnis die Grafiken in Abbildung 3.7. Die Ergebnisse des zweiten Algorithmus finden wir in Abbildung 3.8.
Die Grafiken zeigen, dass wir für unsere Algorithmen auf einem „shared memory“-System den erwarteten optimalen Speedup erreichen. Teilweise erhalten wir sogar einen superlinearen Speedup, was durch die höhere Speicherbandbreite und Cacheeffekten zu erklären ist.

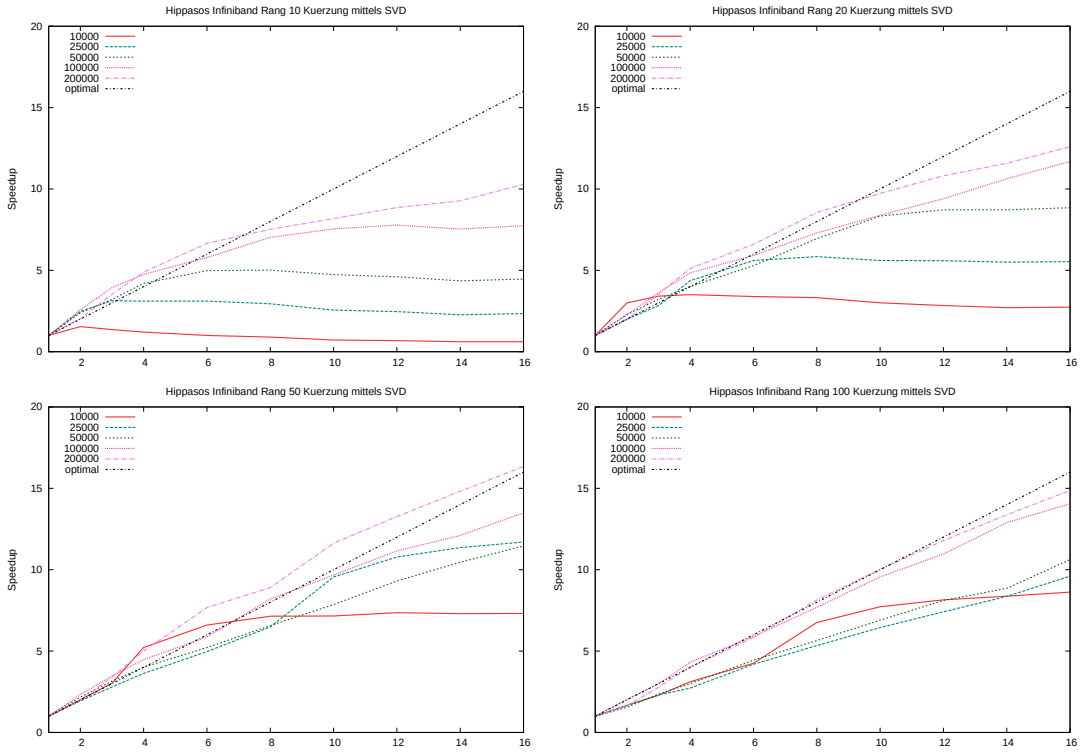


Abbildung 3.5: Test der Kürzung mittels Singulärwertzerlegung mit Infiniband-Netzwerk

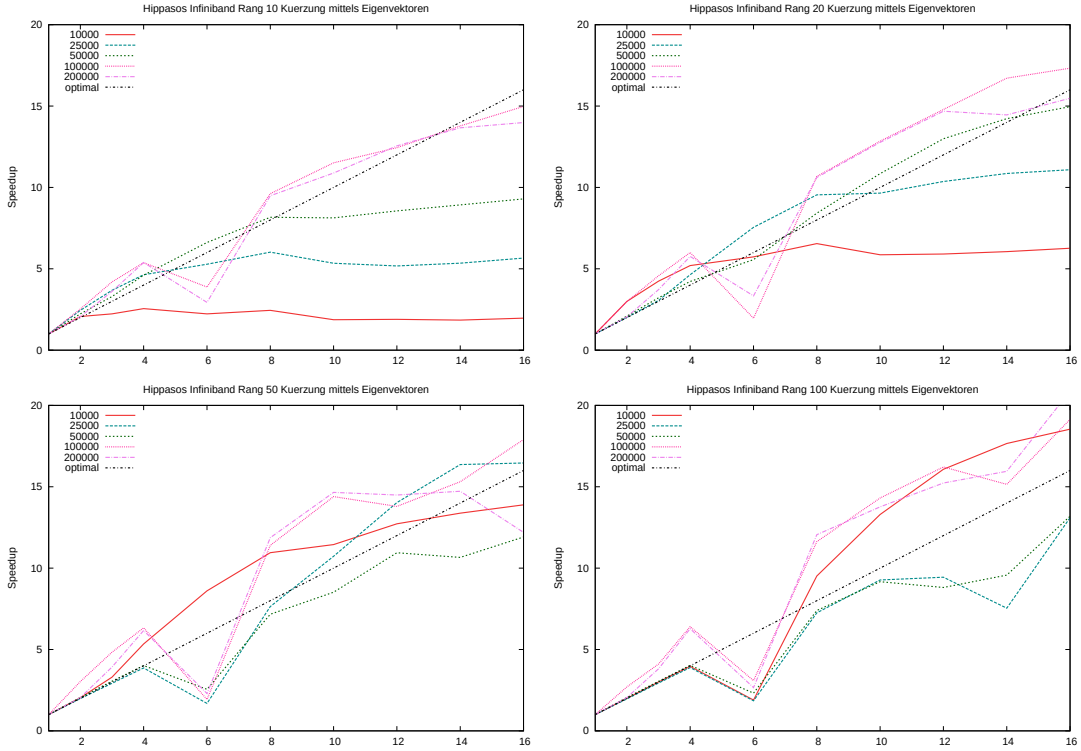


Abbildung 3.6: Test der Kürzung mittels Eigenvektoren mit Infiniband-Netzwerk

Auch für diese Tests gilt wieder, dass der zweite Kürzungsalgorithmus bezüglich des Speedups bessere Ergebnisse liefert. Die Laufzeiten der Algorithmen für 4 Prozessoren sind:

Zeilen	Spalten	Rang	$trsvd$	$trev$
10000	10000	50	0.118s	0.206s
50000	50000	50	0.849s	1.636s
100000	100000	50	1.667s	3.256s

Somit ist auch auf Rechensystemen mit verteiltem Speicher der erste Kürzungsalgorithmus zu bevorzugen, weil er kürzere Laufzeiten hat.

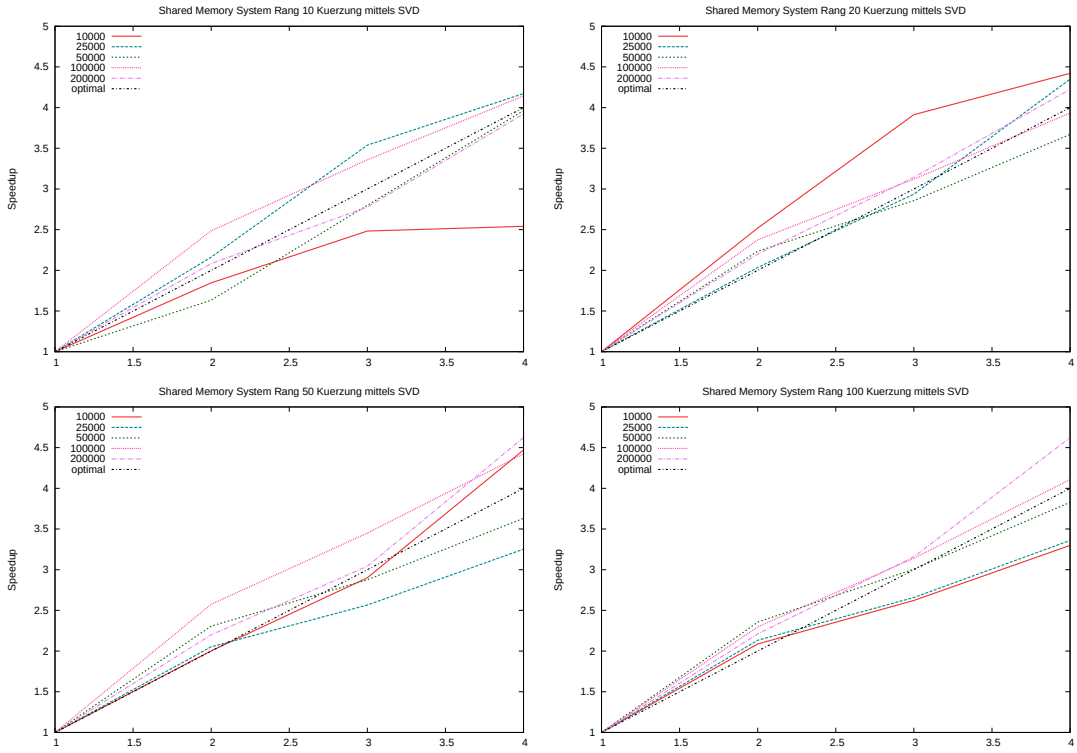


Abbildung 3.7: Test der Kürzung mittels Singulärwertzerlegung auf der Sun-Workstation

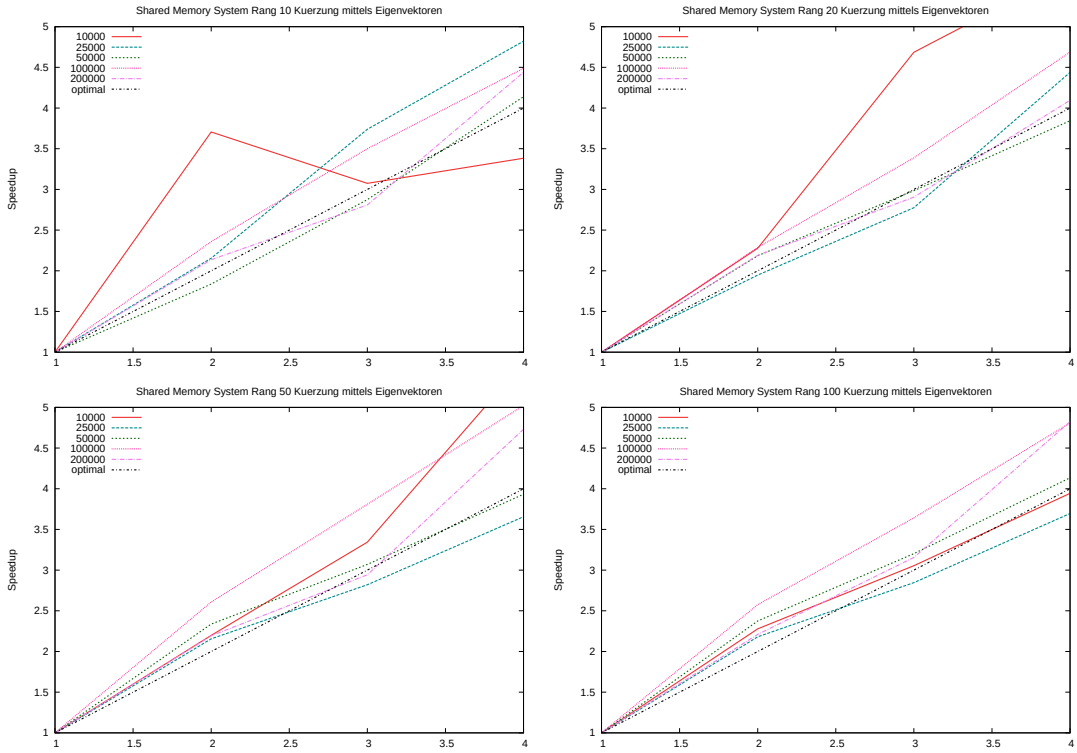


Abbildung 3.8: Test der Kürzung mittels Eigenvektoren auf der Sun-Workstation

Kapitel 4

Fazit und Ausblick

Die Kürzung der Rang-k-Matrizen ist ein wichtiger Baustein der Hierarchischen Matrizen. Wir haben nun für diese Kürzung zwei sequentielle Algorithmen eingeführt und ihre Komplexität analysiert. Dabei haben wir herausgefunden, dass der Kürzungsalgorithmus mittels Singulärwertzerlegung einen geringeren Aufwand hat als der Kürzungsalgorithmus mittels Eigenvektoren.
Dies gilt ebenfalls für die parallelen Versionen der beiden Algorithmen. Die numerischen Tests der beiden parallelen Algorithmen lieferten für den Kürzungsalgorithmus mittels Eigenvektoren eine höheren Speedup. Dennoch ist der parallele Algorithmus mittels Singulärwertzerlegung zu bevorzugen, weil er die geringeren Laufzeiten aufweist.
Des Weiteren konnten die Laufzeitanalyse und die numerischen Tests zeigen, dass wir einen optimalen Speedup für die Algorithmen erreichen können, wenn wir die Anzahl der verwendeten Prozessoren an die Größe der Rang-k-Matrizen anpassen. Eine Hierarchische Matrix besteht nun aus vielen verschiedenen großen Rang-k-Matrizen. Die Herausforderung für die Anwendung der parallelen Kürzungsalgorithmen ist nun eine Hierarchische Matrix so zu verteilen, dass die Kürzung möglichst vieler der Rang-k-Matrizen mit optimalen Speedup durchgeführt werden kann.

Literaturverzeichnis

[1] M. Bebendorf, W. Hackbusch: *Existence of $\mathcal{H}$-Matrix Approximants to the Inverse FE-Matrix of Elliptic Operators with L^∞-Coefficients* Numerische Mathematik, 95:1-28, 2003

[2] J. Bendoraityte: *Parallele Algorithmen zur Matrix-Vektor-Multiplikation mittels $\mathcal{H}^2$-Matrizen* Diplomarbeit Universität Leipzig, 2006

[3] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, P. Spirakis: *BSP versus LogP* Algorithmica(1999) 24: 205-422

[4] R. H. Bisseling: *Parallel Scientific Compuptation - A structured approach using BSP and MPI* Oxford university press, 2004

[5] S. Börm, L. Grasedyck, W. Hackbusch: *Introduction to Hierarchical Matrices with Applications.* Erschienen in Engineering Analysis with Boundary Elements, 27:405-422, 2003

[6] S. Börm, L. Grasedyck, W. Hackbusch: *Hierarchical Matrices.* Zu finden unter: www.hlib.org

[7] E. G. Coffmann Jr., J. R. Lensha, A. H. G. Rinnooy *Computing* North-Holland, 1992

[8] H. Ernst *Grundlagen und Konzepte der Informatik* Vieweg 2000, 2. Auflage

[9] D.E Culler, R. Karp, D. Patterson. A. Sahay, E. Santos, K. E. Schauser, R. Subramonian and T.V. Eicken *LogP: a practical model of parallel computation* Communications of the ACM, 29(11):78-85, 1996

[10] S. Börm, L. Grasedyck: *Low-rank approximation of integral operators by interpolation* Erschienen in Computing, 72:325-332, 2004

[11] S. Börm, L. Grasedyck: *Hybrid Cross Approximation of Integral Operators* Erschienen in Numerische Mathematik, 101, 2(2005) 221-249.

[12] S. Börm, M. Löhndorf, J. M. Melenk, *Approximation of Integral Operators by Variable-Order Interpolation.* Erschienen in Numerische Mathematik, 99, Nr. 4, 2005

[13] D. Braess: *Finite Elemente.* Springer Verlag, 2. Auflage 1997

[14] P. Deuflhard, A. Hohmann, *Numerische Mathematik I - Eine algorithmisch orientierte Einführung.* de Gruyter Verlag, 3. Auflage, 2002

[15] F. Drechsler *Optimierung von Clusterbäumen zu $\mathcal{H}$-Matrixstrukturen* Diplomarbeit August 2006, Uni Leipzig

[16] H. W. Engl: *Integralgleichungen.* Springer Verlag, 1997

[17] G. H. Golub, C. F. Van Loan: *Matrix computations* The Hohns Hopkins University Press, Third Edition, 1996

[18] L. Grasedyck: *Theorie und Anwendungen Hierarchischer Matrizen.* Dissertation, Universität zu Kiel, 2001

[19] L. Grasedyck, W. Hackbusch *Construction and Arithmetics of $\mathcal{H}$-Matrices.* Erschienen in Computing, 70:295-334, 2003

[20] L. Grasedyck: *Adaptive Recompression of $\mathcal{H}$-Matrices for BEM.* Erschienen in Computing, 74:205-223, 2005

[21] W. Hackbusch: *Hierarische Matrizen - Algorithmen und Analysis.* Zu finden unter: http:www.mis.mpg.de/scicomp/Fulltext/hmvorlesung.ps

[22] W. Hackbusch: *Elliptic Differential Equations - Theory and Numerical Treatment.* Springer Verlag, 1992

[23] W. Hackbusch: *Iterative Solution of Large Sparse Systems of Equations.* Springer-Verlag, 1994

[24] W. Hackbusch: *Integral Equations - Theory and Numerical Treatment.* Birkenhäuser Verlag, 1995

[25] R. Kriemann: *Parallele Algorithmen für $\mathcal{H}$-Matrizen* Promotion Universitt Kiel, 2004

[26] B. W. Gernighan, D. M. Ritchie *The C Programming Language* Prentice Hall PTR. Second Edition 1988

[27] LAPACK-Bibliothek und Dokumentation: http://www.netlib.org/lapack/

[28] Dokumentation zum MPI-Standard: http://www.mpi-forum.org/docs/

[29] W. P. Petersen, P. Arbenz: *Introduction to Parallel Computing* Oxford university press, 2004

[30] L. R. Scott, T. Clark, B. Bagheri: *Scientific Parallel Computing* Princeton university press, 2005

[31] O. Steinbach: *Numerische Näherungsverfahren für elliptische Randwertprobleme.* Teubner Verlang, 2003 1. Auflage

[32] E. Tyrtyshnikov, *Mosaic-skeleton approximation* Calcolo 33:47-57, 1996

[33] C. Überhuber, *Computernumerik 2*, Springer Verlag, 1995

[34] L. G. Valiant: *A bridging model for parallel computation* Communications of the ACM, 33(8) 103-11

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort	Datum	Unterschrift